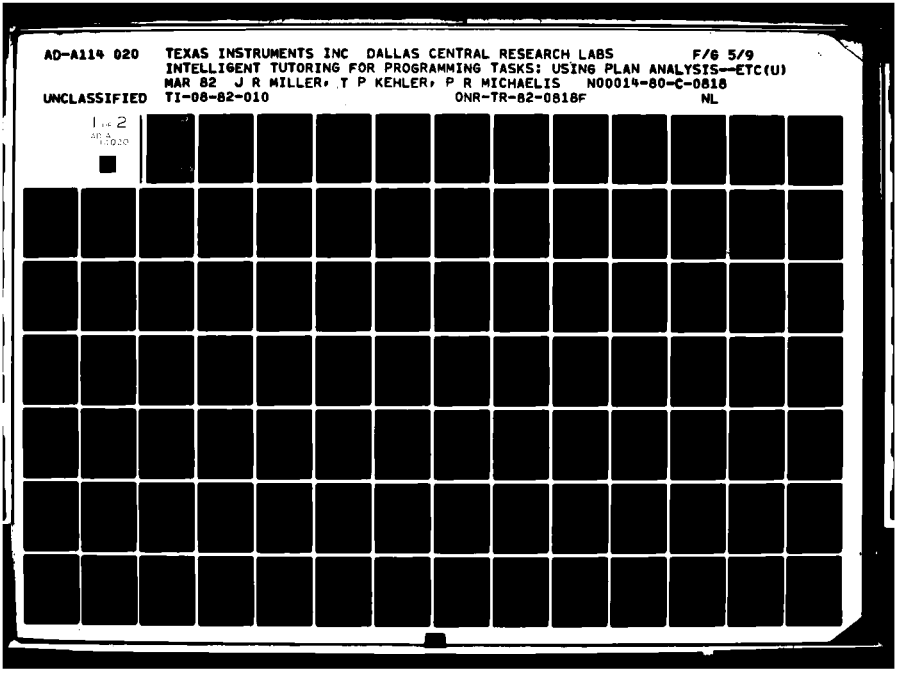

AD-A114 020 TEXAS INSTRUMENTS INC DALLAS CENTRAL RESEARCH LABS F/G 5/9
INTELLIGENT TUTORING FOR PROGRAMMING TASKS: USING PLAN ANALYSIS--ETC(U)
MAR 82 J R MILLER, T P KEHLER, P R MICHAELIS N00014-80-C-0818
UNCLASSIFIED TI-08-82-010 ONR-TR-82-0818F NL
1 of 2
AD A 114020

ONR-TR-82-0818F

INTELLIGENT TUTORING FOR PROGRAMMING TASKS: USING PLAN ANALYSIS TO GENERATE BETTER HINTS

Texas Instruments Incorporated
Central Research Laboratories
13500 North Central Expressway
Dallas, Texas 75265

March 1982

Final Report for Period 30 September 1980 - 31 December 1981

Contract No. N00014-80-C-0818

Research Sponsored by

Personnel and Training Research Programs
Psychological Sciences Division
Office of Naval Research
Under Contract No. N00014-80-C-0818,
Contract Authority Identification No.
NR 154-458

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1 REPORT NUMBER: ONR-TR-82-0818F	2. GOVT ACCESSION NO.: AD-A114 028
	3. RECIPIENT'S CATALOG NUMBER:
4. TITLE (and Subtitle): Intelligent Tutoring for Programming Tasks: Using Plan Analysis to Generate Better Hints	5. TYPE OF REPORT & PERIOD COVERED: Final Report 30 Sept 1980-31 Dec 1981
	6. PERFORMING ORG. REPORT NUMBER: 08-82-010
7. AUTHOR(s): J. R. Miller, T. P. Kehler, P. R. Michaelis, and W. R. Murray	8. CONTRACT OR GRANT NUMBER(s): N00014-80-C-0818
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Texas Instruments Incorporated, Central Research Laboratories, 13500 N. Central Expressway, Dallas, Texas 75265	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research, 800 North Quincy Street, Arlington, VA 22217	12. REPORT DATE: March 1982
	13. NUMBER OF PAGES: 103
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release, distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer-aided instruction
Artificial intelligence
Tutorial systems
Computer programming
Cognitive psychology

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This project has dealt with tutorial systems for computer programming languages, particularly, systems in which a student is trying to write a computer program and can, upon request, receive hints about errors in his program and ways he can correct these errors. This research had two phases: an experimental investigation of the interaction between a student and a hint-giving tutor, and the construction of a tutorial system that identifies the plan underlying a student's program and gives hints that address errors in this plan.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

In the experimental work, an existing programming language tutor (BIP) was modified to allow students to request and receive hints from a human tutor. The protocols generated during the experimental sessions were analyzed by a taxonomic system that described the events that took place at a sufficiently general level to allow comparison of sessions across different subjects or different programming tasks. The actions of the student, the tutor, and the tutorial system itself were classified.

These experiments suggested that:

- Programs are typically written in two stages: (1) layout of the basic design of the program and (2) correction of the statements that instantiate this design.

- Systems that rely on "canned" hints do not provide adequate assistance for the complex problems faced by students. Although BIP was able to give reasonable help on the syntactic form of language statements, it has no facilities for identifying and offering advice on problems with a program's design; in these experiments, students recognized this shortcoming and relied upon assistance from the human tutors.

The simple syntactic help offered by BIP and similar systems needs to be augmented by more powerful techniques that can understand the plan underlying a student's program and offer advice that corresponds to errors in this plan. Our development of TURTLE, a tutor for the "turtle graphics" component of the LOGO programming language, is one step toward this goal. *TURTLE uses artificial intelligence techniques to identify* and offer help on student plans, thereby gaining the ability to diagnose the design of a student's program as well as its code.

TURTLE identifies student plans by working within the limited domain of turtle graphics and by having access to very detailed representations of the problems posed to students. As a result, it can generate the entire set of plausible solutions for a problem. Identifying a student's plan is then reduced to the much simpler matter of matching the student's program to a relatively small set of possible solutions and adopting the the plan that corresponds to the solution offering the best match. Students are encouraged to decompose a more complex problem into a set of smaller problems, each of which is solved by a separate function. By limiting the size of these functions, the combinatorics of solution generation are kept under control, although at the cost of requiring TURTLE to determine how a student has decomposed a problem. This determination is achieved in much the same way as the identification of a simple program's plan: by comparing the student's decomposition to a known set of possible and plausible decompositions.

Differences between the student's proposed solution and the correct solution -- missing or unnecessary statements, or necessary statements with incorrect arguments -- guide TURTLE's hint-generation component. Since the student's plan is known, the functions of the incorrect or missing statements are also known, and hints can be given that reflect errors in the design and the code of the student's program. The individualized and specific attention required and requested by students is thereby made possible.

Table of Contents

Foreword

This is the final report for ONR Contract No. N00014-80-C-0818, "Intelligent Automated Tutors for Instruction in Planning and Computer Programming". The contract monitor for this research was Marshall J. Farr. Mark L. Miller was principal investigator from October 1980 to September 1981, when he left Texas Instruments. Direction for the research was then transferred to Thomas P. Kehler and James R. Miller.

M. Miller was responsible for the contract proposal and the initial orientation of the research. OGOL (Section 2) was designed by M. Miller and William Murray, and implemented by Murray. The BIP/HINT system (Section 3) was implemented (via modifications and extensions to the original BIP code) by Patrick Duff, Murray, and John L. Shelton. The BIP experiments were run by Paul R. Michaelis, Murray, Hendler, and Duff. The taxonomic analysis of the experiments (Section 3) was done by J. Miller, Michaelis, James Hendler, and Duff. Murray designed and implemented TURTLE (Section 4). The final report was coordinated by J. Miller, with contributions from Kehler, Michealis, and Murray (Section 4).

Section 1

Introduction

Recent advances in computer technology have greatly expanded the power and flexibility of computer-aided instruction (CAI) systems. While such systems have existed for many years, the development of artificial intelligence technology has allowed these systems to become increasingly specialized in their control of the educational process and their interaction with students. As a result, these tutors offer a far wider range of diagnostic and tutorial powers than is available in tutoring systems built with traditional computer technology. A general discussion of intelligent computer-aided instruction systems (ICAI) can be found in Clancey, Bennett, and Cohen (1981) and Sleeman and Brown (1981); the current discussion will be limited to CAI systems that tutor students learning computer programming languages.

One of the first programming tutors was BIP (Basic Instructional Program: Barr, Beard, & Atkinson, 1976). BIP offers an environment in which a student can create and debug programs written in the BASIC programming language. A student working with this system is given a series of problems for which he must write programs. He enters the statements of these programs directly into BIP, which checks the syntax of these statements and advises the student of any errors. BIP also provides a set of facilities that allow the student to list, run, erase, and trace the execution of the programs.

For each of the problems given to a student, BIP has a set

of input-output pairs that describe certain pieces of data and the output that should be produced for these data by the student's program. It also has a BASIC program that can generate this output, given the data. The student can run this reference program, enter his own data, and observe the reference program's treatment of these data. When the student has refined his own program to the point where he believes it is correct, he can submit it to BIP for evaluation: BIP runs the student's program with the data in its input-output pairs and compares the output of the student's program to the output that should be produced by these data. The student is informed of any errors in this output and can use the facilities of BIP to debug his program until its output is correct.

BIP has several facilities for offering help to a student during the tutoring session. The student can request a general description of the current problem, which summarizes the general design of the program and the kinds of statements that are necessary to solve the problem. The student can also ask for a "hint," which is usually a more detailed description of a particularly difficult part of this problem. All these descriptions are determined by the instructor who defines the BIP curriculum; BIP merely presents these to the student upon request. Finally, whenever a student enters a statement or a command that is diagnosed by BIP as being incorrect, he is offered help on that statement or command. For example, if a student entered an illegal IF statement as part of his program, BIP would report that the syntax of this statement was incorrect and would offer further help. If the student requested this

advice, he would receive a general description of the form of an IF statement; further requests for help would obtain examples of other incorrect IF statements and, finally, examples of correct IF statements.

The level of assistance BIP can provide to a student is limited. While BIP can offer help on the general form of a particular statement after an incorrect instance of this statement is entered, it can offer only indirect advice, via references to correct and flawed statements of the same type, on why that statement is incorrect and what must be done to correct it. Similarly, BIP can offer little help with statements that are syntactically correct but logically flawed. Although BIP is more than a frame-based tutor, its assistance must be predefined by the course's authors and written in a general form to address general problems that will presumably be of use to as many students as possible. To go beyond these limitations, tutorial systems must incorporate detailed knowledge about the actions the program is intended to carry out, the nature of the programming statements used to build these programs, and high-level knowledge structures that describe the design of a complex program; i.e., the decomposition of a large problem into an organized collection of smaller, manageable problems. As will be discussed later, artificial intelligence techniques offer an appropriate set of tools for this task.

Another feature of BIP that might be improved by artificial intelligent techniques is the selection of the problems that are presented to students. BIP teaches BASIC by requiring the

student to write programs that accomplish increasingly difficult tasks. The complete system contains 91 tasks, although only about 20 of these are ever presented to a single student. The selection of these tasks depends upon BIP's curriculum information network: BIP contains, for each of the problems in the network, a description of (a) the skills (statement types, such as variable assignment, conditional, input/output, etc.) a student must have to solve this problem, (b) the skills the problem is intended to teach, and (c) the skills that are considered irrelevant to the problem. BIP also builds and maintains a model of the student by noting the skills the student has mastered as a result of completing the programming tasks presented to him by BIP. When a student has successfully completed a task (and thus presumably mastered the newly introduced skills in that task), BIP adds the skills that were taught by this task to its model of the student, compares this model to its curriculum network, and presents the student with a task that presupposes a set of skills that is already possessed by the student and that introduces or develops a skill that has not been mastered by the student.

In this way, BIP can tailor its instruction to the progress of individual students, presenting tasks that gradually advance the student toward a complete understanding of BASIC. Note, however, that BIP's lack of knowledge structures that describe the design-level components of a program (which prevents BIP from offering students assistance at this level) also limits its ability to build a detailed and accurate student model: students learning to program are learning software design as well as a

particular programming language, and a sound tutorial system should represent both of these aspects of the students' experience.

These issues have been considered in more recent tutorial systems that have taken advantage of insights from artificial intelligence research. To provide personalized advice about the design of a student's program and about the actions carried out by individual statements, it is necessary to examine the content of the student's partially correct program, to infer the plan underlying this program, and to offer assistance to the student that addresses the differences between this inferred plan and a plan that describes a correct solution to the problem.

One system concerned with inferring student plans to this end was Goldstein's MYCROFT (1975), a system designed to diagnose figures drawn by "turtle graphics" programs written in Logo. A student working with MYCROFT describes the figure he was trying to draw in a "model language"; a stick figure of a man might be described as:

```
MODEL MAN
M1  PARTS HEAD BODY ARMS LEGS
M2  EQUTRI HEAD
M3  LINE BODY
M4  V ARMS, V LEGS
M5  CONNECTED HEAD BODY, CONNECTED BODY ARMS,
    CONNECTED BODY LEGS
M6  BELOW LEGS ARMS, BELOW ARMS HEAD
END
```

This description states that a MAN has a HEAD, BODY, ARMS, and LEGS as its PARTS (M1). The types of these figures are then identified (e.g, the HEAD is an EQUTRI -- an equilateral triangle [M2]), as are the connections between these parts (M5) and the

relative positions of the parts (M6). Some of these property types were defined as system primitives (e.g., PARTS, LINE, CONNECTED, BELOW) while other properties could be defined by the student: here, the student must define EQUTRI and V, in ways similar to the definition of MAN.

MYCROFT could use this figure description to diagnose errors in the student's program in terms of high-level principles underlying the design of the program, such as the failure to reorient the "turtle" drawing device after drawing a portion of the figure, or that an error existed in the statements that drew the LEGS of the MAN. The rules that MYCROFT used to detect programming errors were ad hoc, but captured many of the common flaws in program construction. In addition, MYCROFT was not fully implemented.

Similar concerns for assisting students with the design of their programs lay behind the development of SPADE (Miller, 1979; Miller & Goldstein, 1977). SPADE guided a student through the construction of a Logo program by providing a high-level description of the design of the program that was progressively refined by the student. SPADE used grammar-like structures to represent possible design choices:

```
PLAN      -> IDENTIFY | DECOMPOSE | REFORMULATE
IDENTIFY  -> PRIMITIVE | DEFINED
DECOMPOSE -> CONJUNCTION | REPETITION
...
PRIMITIVE -> VECTOR | ROTATION | PENSTATE
VECTOR    -> {FORWARD | BACK} + <number>
```

A student using SPADE (and the design grammar shown here) is presented with the decomposition of PLAN, indicating that a

program can be written by identifying primitive or previously defined procedures that would accomplish the task, by decomposing the problem into a set of simpler tasks, or by reformulating a problem into a different problem for which a known plan exists. After specifying the desired decomposition, The student continues by selecting one of these decompositions for further work, perhaps DECOMPOSE: by this, the student indicates that the program required a decomposition (although he can later return to the PLAN level and work on the IDENTIFY or REFORMULATE steps, should these be relevant). The student is then shown the decomposition of the DECOMPOSE step and further decides whether the required decomposition is one of CONJUNCTION or REPETITION. As this process continues, the student ultimately reaches the terminal points of the grammar, which specified explicit programming language statements. The result of this attention to the design component of programming is meant to convey basic concepts of software design rather than the syntax of the programming language.

A somewhat different approach to these issues was taken by Gentner's FLOW tutor (Gentner, 1977; Gentner & Norman, 1977). This system contained schematic structures that described a wide variety of states that might be present during the construction of a program, from individual keystrokes to complex design-level structures. The system began its control of a tutoring session by locating the schema that matched the problem posed to a student and expanding this schema into instances of lower-level schemata, predicting individual statements and even keystrokes. The correctness of a student's progress toward a solution of the

problem could then be tracked by noting which schemata were activated by the student's program. If the student proceeded toward a correct solution of the problem, the schemata activated by the student's program would match those activated by the tutor's decomposition of the problem description. Alternative correct solutions to a problem would be correctly interpreted by the system, since a correct solution would satisfy the constraints of high level schemata, even though the individual program statements would have activated an unexpected set of program statement schemata. If the student's program departed significantly from a correct solution, the resulting errors would activate other schemata that were designed to recognize these errors, which would then trigger appropriate corrective actions. It is important to note that these error-detecting schemata could be constructed so as to describe and detect errors at any level of complexity, from an assignment statement with unbalanced parentheses to the incorrect initialization of a counter variable in a complex iterative loop.

The present research has focused on one of these problems -- how to give students specific, individualized assistance during the construction of computer programs. We proposed to construct a tutor that would teach students how to write programs that produce "turtle graphics" line drawings, and that would offer the students hints about errors in their program. The proposed tutor was intended to address not only syntactic errors in program statements, but also errors in the conceptual design of the program. This research was intended to focus on one aspect of the hint-giving problem: the construction and presentation of

relevant hints. We have postponed until a later time the second aspect of this problem -- determining when the tutor should interrupt the student's work to offer a hint -- by requiring the student to explicitly request hints from the tutorial system.

This research is presented in four sections. First, a programming language for studying tutorial interactions and a simple tutor for this language are described. Second, a series of experiments on a computer-based tutor with human-generated hints is presented. Third, the tutoring system that was implemented is described and demonstrated. Fourth, the research is summarized and evaluated, and directions for future research are considered.

Section 2

OGOL: A Graphics-Oriented Language for Tutorial Systems

2.1 Task Definition and Language Construction

The first step in this project was to identify a programming language that would be particularly appropriate for the development of a tutorial system. The primary candidates for this language were Logo and Lisp. Logo-based "turtle graphics" programs are easy to explain to students, can be made arbitrarily complex, and result in a drawing in which the correct and incorrect parts of a program are quite apparent. In addition, these drawings can be represented by networks of points and vectors, a convenient data format for those parts of the tutorial system that must analyze and evaluate student programs. However, the limited power and generality of Logo's iterative and conditional branching statements would make the construction of programs that require these statements inordinately difficult and would limit the extension of a Logo-based tutorial system to other languages in which more powerful control statements are available. In contrast, Lisp offers a straightforward solution to the extensibility problem; however, it lacks the graphics statements that made Logo so appealing and would impose upon the student a set of function names and programming structures that are often difficult and confusing for those who are new to programming or who are experienced with Fortran- or Pascal-like languages.

This dilemma was resolved by developing a new language --

OGOL (ONR Graphics Oriented Language) -- that merges the desirable features of Logo and Lisp. In particular, OGOL allows the construction of turtle graphics programs in an environment with all the extensibility and recursive power of Lisp and with several features that correct some of the more complex and difficult to learn aspects of Lisp: OGOL offers flexible list construction facilities, pattern directed evaluation, and both call-by-name and call-by-value evaluation.

2.2 An OGOL Tutorial System

To evaluate the usefulness of this language, a simple "turtle graphics" tutor was built around OGOL. The Lisp-like features of OGOL were not studied in this system. Rather, the OGOL tutor trains the student in the use of the basic turtle graphics commands. This tutor uses two terminals: a standard terminal with which the student carries on the tutorial interaction with the system, and a color graphics terminal that displays the graphics commands and programs run by the student or the tutorial system. The graphics commands available in OGOL are:

-**ERASE**: erase the screen and place the turtle in the center of the screen.

-**DRAW X**: draw a line X steps long.

-**MOVE X**: move the turtle forward X steps (without drawing a line).

-**TURN X**: turn the turtle X degrees clockwise.

This tutor is in many ways similar to BIP. Like BIP, the OGOL tutor uses a "curriculum information network" to expose a student to the course material in an ordered way. The OGOL tutor

also has a facility for offering predetermined hints about the tasks being presented to the student. The instructor who assembles a particular version of the OGOL tutor can specify a set of hints for each program in the curriculum. These hints can refer to either the general design of the program being written or to the code used in specific statements within the program. Upon request, these hints will be shown one after another in the order specified. The presentation of these hints is partially affected by the student's progress in the task: if the student asks for hints before he has tried to enter a program that would solve the current problem, he will receive the design-level hints provided by the instructor. However, he cannot receive code hints until he has made at least one attempt to define the program.

A partial transcript of a student's interaction with the OGOL tutor is shown in Table 1. The session begins by presenting some introductory information about computer programming in general and turtle graphics programming in particular, and by allowing the student to experiment with the basic OGOL graphics commands -- the student can issue graphics statements and observe their result on the graphics terminal. In the transcript in Table 1, the student has completed this phase of the session and is receiving instruction on how to generate a sequence of OGOL statements that will draw a square.

The student is first told that he can draw a square by putting the individual commands he previously used to draw a square into a program definition statement (line 1). The tutor

Table 1:

A sample tutorial session with the OGOL tutor.
Those lines entered by the student (such as line 15)
are headed by "***".

```
 1 |        Let's write a program to draw a square.  You
 2 |  already know how to do this by typing in a sequence of
 3 |  commands. Write a program named SQUARE that will perform
 4 |  that sequence of commands when it is called.  Make sure
 5 |  that the heading of the turtle before and after calling
 6 |  the SQUARE program is the same.  This may require
 7 |  another TURN command at the end of the sequence of
 8 |  commands to draw the square.  This final turn command
 9 |  will make sure that the sum of all the rotations in the
10 |  SQUARE program is 360.
11 |
12 |  Welcome to OGOL!
13 |  Type (HELP) for help.
14 |
15 |  **(hint)
16 |
17 |  If I wanted to define a program called FOO to have my
18 |  turtle draw forward 100 units, turn 135 degrees, then
19 |  draw 50 units, I would type the following:
20 |
21 |  (define [foo]
22 |      (draw 100)
23 |      (turn 135)
24 |      (draw 50))
25 |
26 |  Note that I can place the various parts of the program on
27 |  different lines.  When all the parentheses balance [that
28 |  is when the number of left parentheses:  ( matches the
29 |  number of right parentheses: ) ] and I have entered a
30 |  return after the last line, then the program is defined.
31 |
32 |  To call FOO, I need only type:
33 |  (FOO).
34 |
35 |  **(hint)
36 |
37 |  Sorry, no more hints are available for designing a
38 |  solution to this task.  However, I can give you more
39 |  hints once you DEFINE a program.
40 |
41 |  **(define [square]
42 |            (draw 100) (draw 100) (draw 100) (draw 100))
43 |  SQUARE
44 |
45 |  **(hint)
46 |
```

```
47 | Your program should call the same sequence of commands
48 | you called before when you drew 4 line segments at right
49 | angles to each other.  If you make an error, you can
50 | always redefine your program and the new definition will
51 | override the old definition.  If your sequence of
52 | commands turns the turtle 270 degrees, then adding (TURN
53 | 90) at the end of that sequence will restore the
54 | original heading of the turtle.
55 |
56 | **(define [square] (draw 100) (turn 90)
57 |                    (draw 100) (turn 90)
58 |                    (draw 100) (turn 90)
59 |                    (draw 100) (turn 90))
60 | SQUARE
61 |
62 | **(square)
63 |
64 | [[ OGOL draws a square on the graphics terminal ]]
65 |
66 | **(quit)
67 |
68 | Here is one way to solve the task you have just
69 | completed:
70 |
71 | (define [square]
72 |         (draw 100)
73 |         (turn 90)
74 |         (draw 100)
75 |         (turn 90)
76 |         (draw 100)
77 |         (turn 90)
78 |         (draw 100)
79 |         (turn 90))
80 |
81 | Note the last (Turn 90) is used to return the turtle to
82 | its original heading.
83 |
84 |    See if you can draw a triangle now, in a similar
85 | fashion to the way you drew the square, using only DRAW
86 | and TURN commands.  The triangle should be equilateral
87 | and have sides of length 100.  It's OK if the triangle
88 | is tilted.
```

[session continues]

then places the student in the OGOL interpreter (line 12), where he is to define the SQUARE function.

The student begins this phase of the session by requesting a hint from the tutor (line 15), which provides a hint about the design of the DEFINE statement: he is shown how to define a function that draws an arbitrary line segment (line 21). The student requests another hint (line 35), but since the student has not yet tried to define the SQUARE function, and since only one design hint was included in the system by the instructor, he is told that no more hints will be available until he has tried to define the function (line 37).

At this point, the student (incorrectly) defines SQUARE (line 41). His subsequent request for a hint is answered (line 47) with a reference to the series of commands he had entered earlier. He then (correctly) redefines SQUARE, (line 56) and runs the function (line 62), which draws an acceptable square on the graphics terminal. Having completed the function, he leaves the OGOL tutor (line 66) and returns to the tutor, which shows him a valid function for SQUARE (line 71). The session then continues, with the student now requested to define a function to draw a triangle (line 94).

2.3 Evaluation

As stated earlier, the OGOL tutor is quite primitive. All hints are prespecified by a human instructor, and its curriculum network was never developed to the extent of the corresponding network in BIP. Further, this system was never extended toward becoming a truly intelligent tutorial system, nor even toward

acquiring the full capabilities of a BIP-like system. As a result of some preliminary experiments with OGOL, it became clear that the basic turtle graphics domain itself -- without OGOL's expanded facilities for subroutine parameter passing and flow of control -- was sufficiently complex for the purposes of the contract. Hence, the development of the OGOL tutor was suspended in favor of developing a turtle graphics system capable of identifying the student's plan and generating intelligent hints about that plan.

Section 3

BIP/HINT: Experiments with a Tutor with Human-Generated Hints

3.1 The Tutorial System

An important part of this study of intelligent tutorial systems has been to study how human tutors interact with their students. Good tutors can build sound models of their students' knowledge and problem solving strategies and can interact with the student in ways that address and correct the difficulties he faced. Identifying the strategies that good human tutors use to these ends is the first step in the construction of tutorial systems with similar capabilities.

To this end -- understanding the relative strengths and weaknesses of human and computer tutors -- experiments were carried out with a modified version of BIP (Barr, et al., 1975). These modifications allowed a student in a tutoring session to request a hint from a human tutor. In this modified version of BIP (here called BIP/HINT) the student and the tutor were located in separate rooms, each equipped with two terminals (see Figure 1). The student used one of the terminals to interact with BIP/HINT; both sides of this interaction also appeared on one of the tutor's terminals. At any time during the session the student could request a hint from the human tutor by pressing the touch panel mounted on his second terminal. The tutor entered this hint on his second terminal, which then appeared on the student's second terminal. This procedure was intended to simulate a tutorial system with a "hint button": in these

experiments, the student had to request a hint (the tutor could not interrupt the student's session and offer a hint), and the student could communicate with the tutor only by pressing the touch panel (the student could not send a question to the tutor).

Some other modifications were made to make BIP more suitable for the current purposes:

* Session recordings. A student's interactions with BIP/HINT and with his tutor were saved in disk files for later analysis.

* Restricted set of tasks. Only seven of BIP's original tasks were used in these experiments:

-GREENFLAG: an introduction to the use of BIP and the format of BASIC statements, and the guided construction and execution of a BASIC program that assigns an integer value to a variable and then prints the value of that integer.

-ARTICHOKE: assign the string "ARTICHOKE" to a string variable, assign the value of that variable to a second variable, and print the second variable.

-SINOP: get two strings from the student with two separate INPUT statements and print them on the same line.

-NINOP: get two numbers from the student and print their sum, difference, product, and quotient.

-TWOS: use a FOR loop to generate and print the even numbers from 2 to some number specified by the student.

-NGREAT: get two numbers from the student and print them in the proper order in the form, "<number> is greater than <number>".

-CALCULATOR: get an arithmetic operation from the student (1="add," 2="subtract," etc.) and apply this operation to two numbers that are then entered by the student.

Some minor changes were made in these tasks to simplify the wordings of the problems and to remove a few ambiguities.

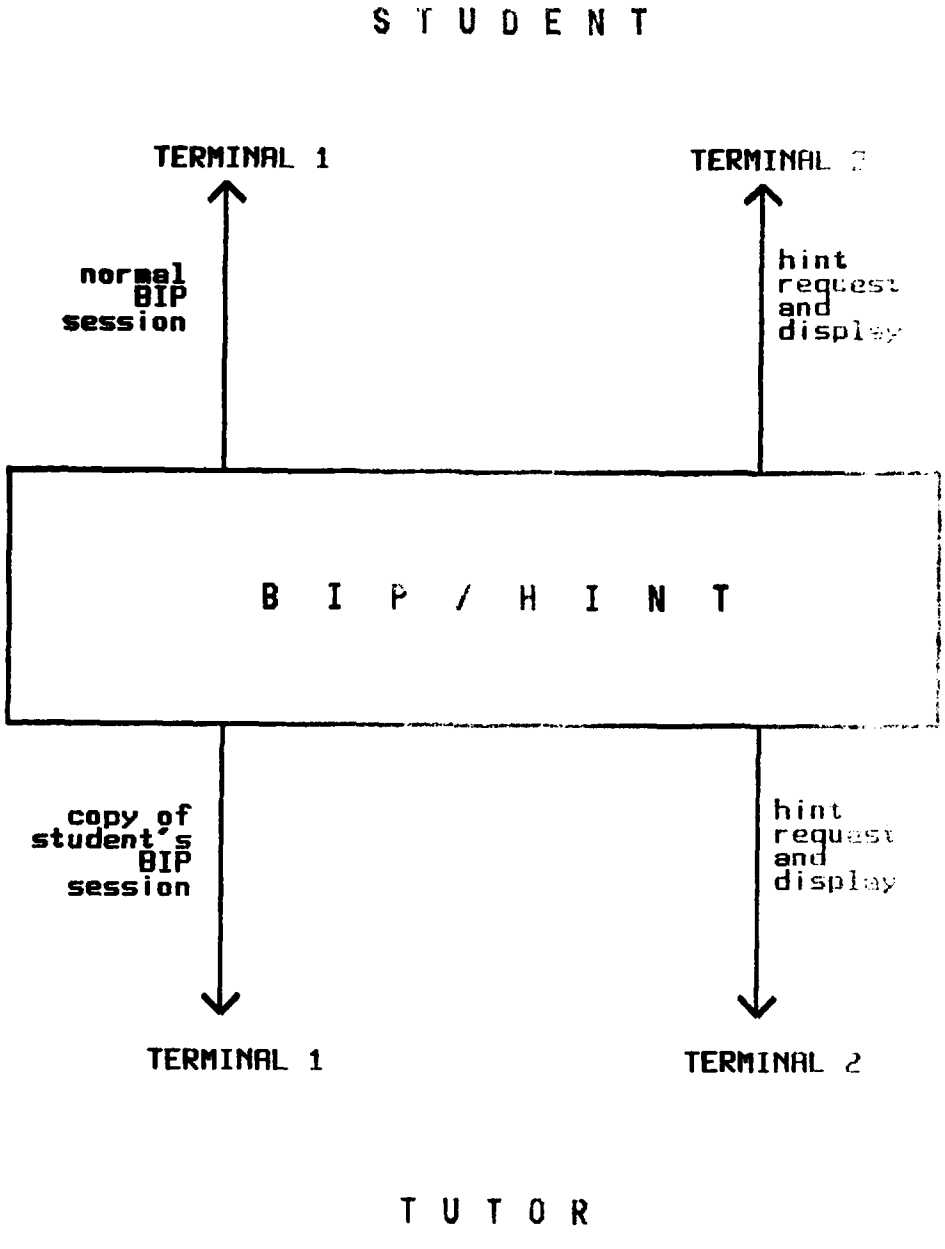


Figure 1 A Schematic Description of the Experimental BIP/HINT System

* Restricted set of BIP commands and BASIC statements. Students working with BIP can be shown the names of the legal BIP commands and BASIC statements by using the commands "?BIP" and "?BASIC", respectively. Both commands produce large amounts of information, and some students complained about being unable to find a command or statement in these lists. Because of this and the restricted set of tasks used in BIP/HINT, some of the commands and statements were removed from BIP/HINT's response to the "?" queries, although the parts of BIP responsible for interpreting the disabled commands and statements remained intact, so that these statements would be interpreted properly if they were entered. The commands and statements available in BIP/HINT were then as follows:

Available BIP commands: CALC, DEMO, LIST, MORE, RUN, SCR, WHAT, WHO, WHY

Hidden BIP commands: DEMO-TRACE, FLOW, HINT, KILL, MORE, BYE

Available BASIC statements: DATA, END, FOR, GOTO, IF, INPUT, LET, NEXT, PRINT, READ, REM, STOP.

Hidden BASIC statements: BEGINSUB, DIM, ENDSUB, GOSUB, REOPEN, RETURN

* Revised student manual. The BIP student manual was revised to make it easier to locate desired parts of the manual and to remove those parts of the manual that referred to BIP commands that were not available in BIP/HINT. This revised manual included selected sections of the original BIP manual and new

one-page summaries of the most common BIP commands and BASIC statements. These summaries contained brief descriptions of the command or statement and the page number of the manual where a more detailed description could be found. This revised manual covered all the legal BASIC statements, but was only 37 pages long (versus the 61 pages of the original manual).

3.2 Experimentation and Protocol Analysis Taxonomy

The experimental sessions were conducted to study the conditions under which students requested assistance from the human tutor or from the tutoring system, the nature of these hints and the tutoring experience in general, and the ability of the student to apply these hints toward the construction of a correct program. There are important differences between BASIC and LOGO, but our analyses of these protocols are not concerned with the language-dependent aspects of the tutoring sessions. Rather, they are focused on the cognitive aspects of tutoring, especially those related to the circumstances surrounding hint requests and hint generation. It should also be noted that these experiments should not be regarded as an evaluation of BIP. The version of BIP used in these experiments was severely limited in comparison to the actual system -- in particular, BIP's own hint facility was hidden from the students. The intent was only to observe the nature of the interaction between a student and a tutor.

Eight subjects, none of whom had any prior experience with computer programming, took part in a series of sessions with BIP/HINT. Due to problems with the computer system (e.g.,

BIP/HINT crashes during a tutorial session), we have usable protocols from three of these subjects. All subjects successfully completed the first five tasks: GREENFLAG, ARTICHOKE, SINOP, NINOP, and TWOS.

Because the protocols are very long, it was necessary to condense them into a form more amenable to analysis. A taxonomy was therefore devised to describe the interactions by the student, the tutor, and BIP/HINT in these protocols (Table 2).

The taxonomy contains three major classifications, corresponding to whether the protocol segment being encoded was produced by the student, by the human tutor, or by the tutorial system. A student may do a variety of things to program statements: he may try to enter a line in the reference program for the first time, reenter that statement until a syntactically legal BASIC statement has been entered, change that legal statement further, or delete the a statement altogether. He may also enter various BIP commands or request help from either the tutor (by pressing the touch panel) or the tutorial system (in response to an offer of help, as described below). The student may also respond to one of a number of questions from BIP.

Our tutors'[1] statements fell into two categories: offers of future help and hints of various kinds. We identified four major classes of hints:

-code: hints that included examples of executable BASIC statements.

---

[1] Several people served as tutors in these experiments.

Table 2:

Taxonomy for the analysis of BIP protocols

```
student
        enters
                command <name <args>>
                data
                statement:
                        line <BIP-line#>
        reenters
                statement:
                        line <BIP-line#>
        changes
                statement:
                        line <BIP-line#>
        deletes
                statement:
                line <BIP-line#>
        evaluates
                program:
                        ok:    {incorrect}
                        wrong: {incorrect}
        answers BIP query <with response>
        asks for
                system
                        {no} help
                tutor
                        hint


tutor
        gives hint:
                code:
                        {ok | error | continue},
                        line <BIP-line#>
                design:
                        {ok | error | continue},
                        line <BIP-line#>
                operation:
                        {ok | error | continue}
                general:
                        {ok | error | continue}
        offers
                future help
```

```
system
        accepts
                command <name>
                modification: line <BIP-line#>
                statement: line <BIP-line#>
        asks for operation instructions
        describes execution:
                fails: line <BIP-line>, error <BIP-error>
                input error
                runs
                        ok
                        wrong
        gives hint
                code:
                        line <BIP-line#>,
                                continue
                                error
                                ok
                design:
                        line <BIP-line#>,
                                continue
                                error
                                ok
                operation:
                        continue
                        error
                        ok
                general:
                        continue
                        error
                        ok
        offers
                help:
                        line <BIP-line#>
                        operation
                no help
                problem description
        rejects
                command: <name <args>>
                        has a line number
                        issued at wrong time
                        incomplete command
                        other error
                statement: line <BIP-line#>,
                        has no line number
                        incomplete-statement
                        other error
                        syntax error: <BIP-error>
        terminates task
```

-design: general hints about the design of the program, or hints that suggested the use of particular BASIC statements, but that did not show an executable statement. Hence, the hint, "Use a PRINT statement to output the value of X" would be scored as a design hint, while "Use the statement 'PRINT X' to output the value of X" would be scored as a code hint.

-operation: hints about the operation of BIP (e.g., "Use the LIST command to look at your program.").

-general: hints that referred to general aspects of computer programming (e.g., "The computer can only do exactly what you tell it to do.")

In addition, the scoring of the hint noted the line of the reference program with which the hint was concerned, and whether, through this hint, the tutor was telling the student that an error had been made, that the relevant part of the program was correct (ok), or that the student should simply continue working.

The parts of the protocol generated by the tutorial system were scored in similar ways. The system's acceptance or rejection of the BASIC statements and BIP commands entered by a student were scored appropriately. BIP could also offer help after a statement or command error, ask the student for certain operation instructions, and describe the execution of the student's program. These diagnostic messages and problem descriptions typically contained suggestions about how the program should be constructed, and so were scored the same way as were tutor hints.

This taxonomic system converts the near-natural language of the tutorial session into a more limited form, which offers several advantages now and others that might be explored in the future. The use of this taxonomy has allowed us to automate much

of the descriptive analysis of these protocols, and it serves as a common language through which we can compare the performance of all the subjects on a common task, or evaluate the progress of individual subjects on successive tasks. A sample protocol and its taxonomic analysis are shown in Table 3. Statements from BIP are in upper case, responses from the student are in lower case; the taxonomic analyses of the statements follow exclamation points, and the tutor's hints are underlined.

The session shown in this protocol began with an offer by BIP to print a description of the current task (lines 1-2); this was coded as "system offers problem description" (line 3). By responding "yes" (line 5), the "student answers [a] BIP query." This description of the problem contained several hints of different kinds. First, the student was shown two explicit examples of how variables can be assigned the values of string constants or other string variables (lines 9 and 15). Since these "system hints" contained instances of executable BASIC code, they were further classified as code hints about specific lines of BIP's reference program for this problem.[2] The student also received other, more abstract hints about the design of this particular program; these (lines 19, 23, and 26) were textual descriptions of what each line of this program should do, and

---

[2] The BIP tasks we have studied have been simple enough that our students' programs rarely differed in any substantial ways from these reference programs. As a result, we were able to use BIP's reference program as a standard against which the students' programs were scored: the program statements entered by students were encoded in terms of the corresponding statement in BIP's reference program. This use of a standard reference program would probably not be satisfactory with more complex programs.

Table 3:

A sample application of the taxonomy to a BIP protocol. BIP's exchanges are in upper case, the student's responses are in lower case, the tutor's hints are underlined, and the taxonomic statements are to the right of an exclamation point.

```
 1 | YOU ARE IN TASK ARTICHOKE.
 2 | DO YOU WANT THE TEXT PRINTED OUT?  TYPE Y OR N.
 3 |                              ! (system offers problem
 4 |                              !       description)
 5 |  >y
 6 |                              ! (student answers BIP query: y)
 7 |
 8 | YOU CAN ASSIGN A LITERAL VALUE TO A VARIABLE LIKE THIS:
 9 |         10 T$ = "TELEPHONE"
10 |                              ! (system gives hint:
11 |                              !       code: line 10)
12 |
13 | YOU CAN ALSO ASSIGN THE VALUE OF A VARIABLE TO ANOTHER
14 | VARIABLE, LIKE THIS:
15 |         10 S$ = T$
16 |                              ! (system gives hint:
17 |                              !       code: line 20)
18 |
19 | WRITE A PROGRAM THAT FIRST ASSIGNS THE VALUE "ARTICHOKE"
20 | TO THE STRING VARIABLE A$.
21 |                              ! (system gives hint:
22 |                              !       design: line 10)
23 | THEN HAVE IT ASSIGN THE VALUE OF A$ TO ANOTHER VARIABLE B$
24 |                              ! (system gives hint:
25 |                              !       design: line 20)
26 | AND THEN PRINT THE VALUE OF B$.
27 |                              ! (system gives hint:
28 |                              !       design: line 30)
29 |
30 | *10 "artichoke" a$="artichoke
31 |                              ! (student enters statement:
32 |                              !       line 10)
33 |     ^
34 | SYNTAX ERROR: UNMATCHED QUOTE MARKS
35 |                              ! (system rejects statement:
36 |                              !       line 10:
37 |                              !       syntax_error)
38 | LINE NOT ACCEPTED.  (TYPE ? FOR HELP)
39 |                              ! (system offers help)
40 |
41 | *10 artichoke                ! (student re-enters statement:
42 |                              !       line 10)
43 |
44 |                              ! (student requests hint)
45 |
46 | From TUTOR: you're doing it backwards!
47 |                              ! (tutor gives hint:
```

```
                                !       design, line 10)
Try typing the line in this format:

10 T$ = "TELEPHONE"
                                ! (tutor gives hint:
                                !       code, line 10)

10 a$ ="artichoke
                                ! (student re-enters statement:
                                !       line 10)
                    ^
SYNTAX ERROR: UNMATCHED QUOTE MARKS
                                ! (system rejects statement:
                                !       line 10,
                                !       syntax_error)
LINE NOT ACCEPTED.  (TYPE ? FOR HELP)
                                ! (system offers help)
*what
                                ! (student enters command)

YOU ARE IN TASK ARTICHOKE.
DO YOU WANT THE TEXT PRINTED OUT?  TYPE Y OR N.
                                ! (system offers problem
                                !       description)
 >v
                                ! (student requests system help)
```

[protocol continues]

were scored as design hints.

In response to these hints, the student tried to enter the first line of the program (line 30), which was scored (line 31) as the entry of line 10 of BIP's reference program. The syntax error in this line was scored in BIP's response to the student's input (line 34). BIP then offered help (line 38), which the student did not request. Instead, she made another attempt at the statement (line 41), which was again flawed by a syntax error. Before actually entering this line into BIP, however, she requested a hint from the tutor (line 44) by pressing the touch panel.

In this interaction with the student, the tutor provided two hints of differing specificity: a design hint about the assignment statement (line 46) and a code hint about the correct form of the assignment statement (line 51). The student then tried to enter the correct line (line 55), but made another syntax error. The student again rejected BIP's offer of help (line 63), and entered the command that will print the description of the task (WHAT: line 65). As the protocol ended, she asked to see the task description again (line 72), in which she received the design and code hints that she received in the first presentation of the problem description (lines 8-28), and continued to work on the problem.

This analysis provided the ability to describe a tutorial session at an abstract, but still informative, level. The student's problems were centered around the assignment statement in line 10, having entered attempts at the statement three times

and having received code and design hints about the statement from two different sources. More importantly, the abstract form of this analysis allowed us to evaluate the performance of multiple subjects on these problems, and it is to these kinds of analyses we should now turn.

3.3 Taxonomic Analysis of the BIP Sessions

Even with the limited amount of data we considered, the taxonomic analysis of these protocols revealed several interesting trends. Perhaps the most informative analyses were those in which the tutorial sessions were divided in half,[3] and separate analyses were carried out on each half. In this way, we could observe changes in the performance of the student, the system, or the tutor as the student gradually approached a correct solution.

3.3.1 Statement Composition and Debugging

With the analysis of these protocols it was possible to address the way students construct a program. Recall that our classifications of statement _entry_ and _reentry_ correspond to the initial attempts to enter a syntactically legal statement; any modifications to a legal statement are then scored as _changes_. The mean frequencies of these classes of events for our subjects in the first and second halves of the protocols are shown in Table 4. These data suggest two major processes in our subjects' solution of these problems: statement _composition_ -- in which a rough approximation to the program is constructed -- and

---

[3] On the basis of the number of lines of text in the protocol

statement debugging -- in which the nearly correct components of the proposed solution are corrected and refined.

These processes take place neither strictly sequentially nor strictly simultaneously. Statement composition appears to take place primarily in the first part of the solution; our students made somewhat more enters and reenters in the first half of the protocol. In contrast, statement debugging was concentrated in the second half of the solution: more than twice as many statements were changed in the second half than in the first. Subjects seem to have first made a general pass over the problem, generating a statement (that is not necessarily correct) for each part of the problem, and then gradually refining these statements into their correct forms. This is similar to Atwood and Jeffries's (1980) finding that people approach complex software design tasks by first building, in a breadth-first manner, an approximation to a complete solution, and then refining in a depth-first way those components of the solution that are in error.

3.3.2 Students' Requests for Assistance

Students in these sessions could get assistance from either the human tutor (by pressing the touch panel and requesting a hint) or by asking BIP for help (the diagnosis of an incorrect statement could, if the student wished, be followed by descriptions of the correct and incorrect forms of those statements). Table 5 shows the mean frequencies of subjects' requests for assistance from either the human tutor or BIP itself during the first and second halves of the tutorial session.

Table 4:

Mean frequencies of statement entries and reentries vs statement changes in the first and second halves of protocols.

	entries and reentries	changes
first half	4.8	3.0
second half	2.4	6.2

Table 5:

Mean requests for assistance from the human tutor and the tutorial system in the first and second halves of protocols.

	tutor	system
first half	1.5	0.8
second half	2.1	0.5

These data show that subjects strongly preferred assistance from the human tutor, particularly in the second half of the session, where, as found in the first analysis, subjects were debugging their initial rough approximation to their program. Together, these analyses indicate that our subjects were willing to accept BIP's help when they were laying out the initial plan for the program, but that they generally relied on the human tutor for assistance when debugging their programs. These preferences correspond to the different kinds of help that are available from these two sources: BIP's system help is good for general comments about the structure of program statements and the basic design that the program should take, while the human tutor is most valuable when the student has entered a statement that is an incorrect, but syntactically legal, piece of BASIC code. The human tutor can identify the student's problem and guide the student toward a correct solution.

These data illustrate a further point about our students' use of the system and tutor help facilities -- they were extremely hesitant to take advantage of them. Requests for system help were very rare (especially so when it is remembered that BIP offers this help after every input error made by the student); requests for tutor hints were more common, but still infrequent. The implications of this finding on the design of intelligent tutors will be discussed later.

3.3.3 Problem Difficulty

Differences in the nature and level of help available from the tutors and from BIP can also be seen in Table 6, which shows

the number of times that students requested help from either the tutor or from BIP in each of the four tasks. These data indicate that students preferred help from the tutor at two particular times: when they were just beginning to use the system, and when they were working on the hardest of the four problems.

Subjects need especially flexible and individualized assistance at both of these times. When working on the first problem, subjects needed help on their use of the BIP system itself, particularly which commands should be used at which times. Although BIP contains thorough descriptions of its commands that could be offered when a student incorrectly used the command, it is unable to infer from a student's performance which of these commands might be appropriate at a given time; therefore, its help facilities are of little use at this point. Further, since help is available in this system only after an error has been made, a student may have no choice but to ask the tutor for help in what might be done next.

The final and most difficult problem studied (TWOS) requires the use of the BASIC iteration statement (FOR). To solve this problem correctly, the student must understand the use of iteration in the design of the program: the ability to write the statement undergoing the iteration is assumed, and the skill being learned is how to embed one conceptual part of a program

Table 6:

Mean requests for assistance from the human tutor and the tutorial system for the four completed problems.

	tutor	system
ARTICHOKE	2.0	0.0
SINOP	2.7	2.3
NINOP	2.0	2.3
TWOS	8.7	.7

[Note: these data are ranked by the difficulty of the problems; ARTICHOKE was the easiest of the four; TWOS was the hardest.]

within another.[4] We suspect that our students used BIP's help facilities on the simple problems because the level of help that BIP can offer during these problems matches the level that is needed -- advice on the format of simple input/output and assignment statements. However, to effectively advise a student on a program containing iteration, the tutor must understand the proper relation between the iteration statement and the iterated statement, as well as the student's (possibly flawed) understanding of this relation. This detailed understanding of a conceivably large set of program statements is beyond the scope of a system like BIP.

3.3.4 Evaluation of the Taxonomic System

As an experiment to determine whether such a taxonomic system might be useful in the study of tutoring protocols, the present analyses were quite promising. However, further research involving more subjects and tasks is needed to refine this technique. One direction in which future research should go is toward the generation of the protocol analysis by the tutorial system itself: when a student entered a statement, the system could easily enter the taxonomic statement "student enters statement" into its recording of the tutorial session; the

---

[4] A strong causal statement regarding why students stop requesting hints on hard problems cannot be made from these data, since the difficulty of the problems received by a student is confounded with number of problems that a student has solved. While we believe that the "canned" hints like those offered by BIP are initially useful, but become less useful as the problems become more complex, it may also be that canned hints are never really useful to the students, but that the students must work with BIP for some time before they realize this.

system's acceptance or rejection and diagnosis of that statement could similarly be noted. However, some decisions cannot be made as easily as these simple notations of syntax errors: if a student entered a poorly constructed IF statement during the composition of a program that contained two IF statements, it would probably difficult to determine which of the two the student was trying to write. At this point the taxonomic analysis of protocols would begin to change from a simple transcription of the "surface structure" of a program into a more conceptual analysis of the student's plan that underlies the generated program -- exactly the kind of analysis that is needed in an "intelligent" tutor. It is unclear at this time whether a taxonomic system like that used here might also serve as an abstract descriptive language that could be exploited by the plan identification component of an intelligent tutoring system (cf. Miller & Goldstein, 1977), but this possibility may be worth exploring.

3.4 Detailed Analysis of a BIP Session

Although much can be learned from the taxonomic analyses described above, a more detailed understanding of the tutorial interaction still requires analysis of the original protocols. One of our subjects -- referred to as DD -- found the programming tasks particularly difficult, and was the focus of such an analysis. Her protocol offers offers a particularly rich source of data on the interaction between a tutor and a student.

DD correctly completed the GREENFLAG, ARTICHOKE, SINOP, and NINOP tasks, and, in doing so, demonstrated knowledge of the

following aspects of programming in BASIC:

-Numerical and string constants

-Assignment of a string or numerical constant to a variable (i.e., A=5 or A="ARTICHOKE")

-Assignment of a variable's value to another variable (i.e., A=B)

-INPUT and PRINT statements

DD then moved on to the TWOS task, which is described by BIP as follows:

Write a program that counts by twos, up to a number given by the user. For example, if they gave 8, your program would print:
2
4
6
8

Use a FOR . . NEXT loop for this problem.

This problem can be solved by the following BASIC program:

```
1  REM N IS: THE NUMBER TO COUNT UP TO
10 PRINT "HOW HIGH SHOULD I COUNT?"
20 INPUT N
30 FOR I = 2 TO N STEP 2
40   PRINT I
50 NEXT I
99 END
```

The new skill introduced in this task is the use of the FOR statement for the construction of iterative loops. This task was selected by BIP's curriculum information network because DD had successfully completed tasks that required using input/output statements, the only other statements needed to solve this problem.

DD worked on this task for almost two hours, but never wrote a completely successful program. Her nearly complete program

failed in a way that caused a fatal error condition in BIP which terminated the session.[5] The analysis of DD's protocol revealed some serious confusions about BASIC and computer programming in general:

* She was confused about variable assignment. Her protocol suggests that she initially thought that the INPUT statement should be used to assign values to variables (i.e., "INPUT X 8" means "set X to 8"), and that she did not really understand that using an INPUT statement in her program would allow her to enter a value when she ran the program at a later time. It is important to note that she was confused in this way even though she had successfully completed several other BIP tasks that required her to use INPUT and assignment statements.

* DD's handling of the FOR statement required by this program suggested major misunderstandings at several different levels. She tried several different (and illegal) forms of the FOR statement (such as "FOR X = 2 - 8"), had problems determining the exact relation between the FOR and NEXT statements, and was unsure of the proper way to incorporate the STEP information into the FOR statement. Beyond the syntax of the FOR statement, she had trouble grasping two ideas that are concerned with higher-level aspects of programming: (a) embedding one conceptual group of statements (the statements to be repeatedly

---

[5] DD's program consisted of the statements shown in the sample program, but her specification of the upper bound and the index variable of the FOR was incorrect. It is unclear how much additional work by DD would have been required to correct this error.

executed) within another group of statements (the FOR and NEXT statements that controlled the iteration), and (b) using the FOR's index variable for some purpose in the statements within the iterative loop. Although she finally wrote a program that had the proper collection of statements in the right order -- INPUT, FOR, PRINT, NEXT, and END -- she confused the uses and functions of the FOR's index variable and the number, entered through the INPUT statement, that controlled the upper bound of the loop.

* She never had a good understanding of the overall design of the program. The use of a variable to specify the upper bound of the loop was the last part of the program to be written; throughout the majority of the session, she wrote FOR statements with explicit upper bounds, either 8 or 10. She used the MORE command -- the command that tells BIP that the student has finished working on one problem and wants to go on to another -- several times with programs that contained such an explicit upper bound, indicating that she thought she had a correct program even with that incorrect construction.

Not all these problems can be blamed on DD herself. Rather, some of her problems can be attributed to various aspects of the human tutor, the modified version of BIP, and BASIC.

3.4.1 Problems with BASIC

BASIC is an increasingly common first computer language, but it has many disadvantageous aspects for beginning programmers. Many of these are related to notions of structured programming. Programs written with BASIC's extremely limited control

structures can be very difficult to read and understand. In addition, the restricted length of BASIC variable names (which makes the use of meaningful mnemonics difficult) and the strict format of statements (a line must contain exactly one complete statement) can also be expected to complicate the programming task.

3.4.2 Problems with BIP

There are two aspects of BIP that should be improved in future tutorial systems:

1: Limited display. The student's interaction with BIP must take place within the twenty-four lines of a computer terminal's screen. This greatly restricts the student's ability to work simultaneously with a tutor, his program, and BIP. Twenty-four lines are not enough to allow a student to keep track of the current state of his program, the output produced by that program, recent changes made to the program, interactions with BIP, and hints obtained from several possible sources. DD often entered program lines that were already present (at one point, DD's TWOS program had four END statements), entered new lines with the same statement number as already-entered statements (thereby erasing the old and possibly correct statements), and ran the program with certain statements missing. A system that used multiple "windows" to display the various components of the tutorial interaction would avoid many of these problems. Such terminals were not available when BIP was built, but future tutor projects should take advantage of these systems.

2: Rigid input format. BIP offers no provisions for

correcting spelling or typing errors. Hence, when DD typed "RU N" instead of "RUN", BIP could do nothing more than identify it as an illegal command. If tutorial systems for beginning programming require the students to enter their programs on a keyboard (rather than choosing command names and arguments from a menu, for instance), automatic spelling correction might be useful.

3.4.3 Problems with the Tutor

Certain problems in DD's performance can also be traced to the tutor. In general, our tutors -- skilled computer programmers who generally did not have substantial teaching experience -- did quite well. When a hint was requested, they were consistently able to identify the student's problem and offer a relevant hint. Whether or not that hint was the "right" or the "best" hint is beyond the scope of this research. For now, the best evidence for the appropriateness of our tutors' hints is that, after receiving hints, students always worked on their problem before requesting another hint; on no occasion did a student indicate, by asking for two consecutive hints, "I understand that -- my problem is...".

Our tutors' problems stem primarily from the fact that the tutorial domain is very open-ended, and that instantaneously generating high-quality hints about any aspect of a programming problem and a student's solution to that problem is not easy. Some of these problems were mechanical and avoidable -- occasionally, a tutor would start to type a hint, change his mind, and erase what he had written, one character at a time.

All of this was visible to the student in these experiments; a minor modification would allow the tutor to compose his hint carefully and send it to the student only when he was satisfied with its form.

More relevant to this discussion, however, are the occasions when the tutors' hints were inconsistent or ambiguous. For instance, when DD was trying to construct a legal FOR statement, the following interaction took place:

a: DD requested a hint, with her program in the following state:

```
10 FOR X = 2 TO 8 STEP 2
20 PRINT X
30 END
```

b: The tutor said "You have a FOR on line 10, but nowhere do you have a NEXT."

c: DD tried to change line 10 to:

```
10 FOR X = 2 TO 8 NEXT STEP 2
```

When BIP rejected this illegal statement, DD asked for another hint.

d: The tutor said "I'm sorry, you misunderstood my last hint. You need a NEXT statement on a different line AFTER the PRINT statement...."

The source of this error lies in the differing knowledge structures possessed by the student and tutor that guide their individual comprehension of the sentence. While it is clear to the tutor that NEXT belongs on a separate line -- because of his

prior experience with BASIC -- it is not so clear to DD, and the observed misinterpretation occurs. This incident might have been avoided if the tutor had said in his first hint, "...nowhere do you have a NEXT statement."

A related problem with the different knowledge structures possessed by student and tutor was illustrated by another part of DD's session. DD had requested a hint when she was having problems writing a correct FOR statement; the tutor offered the following hint:

> If we want to set A to go from 1 to 10 and print A we would write the following program:
>
> 10 FOR A=1 TO 10
> 20 PRINT A
> 30 NEXT A
>
> This is similar to your task, except that
> (a) you need to go from 2 to whatever number the user typed.
> (b) you want to count by twos.
>
> This is done by saying:
> FOR variable = start TO finish STEP 2.

DD responded to this hint by entering the statement:

FOR VARIABLE=START TO FINISH

The problem exposed here is that the tutor is using a particular notational system common to experienced computer programmers: statements are described by presenting their required keywords in capital letters (such as FOR and TO), and presenting variables that must be specified by the programmer in lower case terms that describe their function (e.g., variable, start, and finish). DD does not possess this structure, so she interprets the tutor's hint too literally. Mismatches of this nature were common in our protocols; at one point, the tutor was explaining the use of the

STEP keyword in the FOR statement:

> To tell the loop what to step by use the following form:
>
> 10 FOR something = something TO something STEP NUMBER
>
> where NUMBER is the number you want to step by.
> To count by 5's you would say ... STEP 5.

DD then entered:

```
                    6
10 FOR X=2 TO 8
20 STEP 2
```

The mismatch here lies in the tutor's use of "..." to stand for the part of the FOR statement she had already written. In addition, the problem exists because of BASIC's requirement that the STEP information be on the same line as the FOR statement; a less rigidly formatted language would not have posed this problem.[7]

These problems illustrate the advantages and disadvantages of "canned" hint systems and intelligent tutoring systems. While it is impossible to anticipate every occasion for which a canned hint should be constructed, hints can be very carefully worded for those circumstances that can be anticipated. In contrast, while human tutors possess the problem-solving power systems such as BIP lack, the task of diagnosing a student's problem, deriving

---

[6] Note DD's failure to use a variable for the loop's upper bound.

[7] On the other hand, such a language would not have revealed this problem in DD's conceptualization of the task, and DD would probably have written her future FOR loops with the STEP information on a separate line.

a sound hint for that problem, and converting that hint to an unambiguous piece of natural language (all while being under the time pressure of trying to generate this hint as quickly as possible) is extremely complex for even human tutors.

3.4.4 Problems with the "Hint Button" Approach

We originally proposed to study tutoring systems with "hint buttons" as a way of isolating one part of the intelligent tutoring domain. By providing hints only upon request, we could bypass the complex issue of identifying the circumstances under which the tutoring system should interrupt the student with a hint, and concentrate our efforts on techniques for generating these hints. These techniques are described in the discussion of TURTLE (Section 4). However, with the present experiments, we can address the question of whether successful tutors might be built by taking this shortcut, and, like the modified BIP system and TURTLE, giving hints only when a student requested them. These experiments suggest that such a shortcut would not be reasonable.

A surprising finding of these experiments was that our subjects were hesitant to request hints from the tutor; requests averaged only about four per session (Table 6). The fact that, in these experiments, the students had to ask another person for help does not seem to be part of the problem. The students were also unlikely to take advantage of BIP's help facilities, and, in some informal experiments, there was a similar hesitancy to request hints on the part of students who worked with TURTLE, whose hint facility was completely automated.

Our experiences with both tutorial systems suggest that one of the commonly cited educational advantages of programming -- that writing a computer program is often like solving a puzzle or even playing a game -- works against the idea of a "hint button". Subjects seemed to treat the use of the hint button as cheating, or something that should be done only as a method of last resort.

It could be argued that our subjects were simply not motivated strongly enough to request hints, and that more encouragement would lead to more requests for hints. However, this would ignore the long-range problem of the design of an "intelligent", hint-giving, tutor. One of the reasons our tutors could successfully identify the student's problem when a hint was requested was because students requested hints so infrequently. When DD entered

```
10 FOR X = 2 TO 8
20 STEP 2                {flagged as illegal by BIP}
20 STEP 2                {flagged as illegal by BIP}
20 X = STEP 2            {flagged as illegal by BIP}
STEP 2                   {flagged as illegal by BIP}
20 STEP 2                {flagged as illegal by BIP}
20 STEP 2                {flagged as illegal by BIP}
```

before finally asking for a hint, it was clear she was having trouble with the syntax of the FOR/STEP statement. Without this repetition, the tutor might not have been able to identify her problem. Suppose DD had entered line 10, as in the above example, and, after realizing that she did not know how to specify the STEP statement, asked for a hint. Working only with the single statement thus far entered by the student, the tutor could offer hints referring to the absence of of the STEP component of the FOR statement, the missing NEXT statement, or the fact that the FOR uses an explicit upper bound rather than a

variable. Only one of these alternatives would correspond to the student's real problem, and the tutor would need additional evidence to select the right one. With frequent hints and no supplementary information from the student to help identify his problem, effective tutoring would be difficult.

The study of these protocols leads to two conclusions. First, interruption by the tutorial system to give certain hints would not be beyond the capability of current "intelligent tutoring systems" technology. Many of the hints students requested were preceded by repeated syntax errors like those shown above. While the comprehension and diagnosis of a computer program's design is considerably more difficult than identifying syntax errors, a tutorial system capable of diagnosing repeated syntax errors and interrupting the student with a hint about the statement's proper form would not be difficult. Further, a detailed theory of how people learn to program is not really needed in order to suggest that a student who makes several consecutive identical errors should be interrupted with corrective advice (although such a theory would be critical to generating the best possible hint).

Second, the communication between a student and a tutorial system should be more flexible than allowing the student, in effect, to say nothing more than "I need help!". In most cases, the diagnosis of a student's problem will be facilitated by allowing the student to describe why he is requesting a hint. Natural language would ultimately be useful for this purpose; although the difficulties of building natural language systems in

unconstrained domains are well-known, a tutorial dialogue in a particular domain might provide sufficient constraint to make such a system possible (cf. Brown & Burton, 1975; Stevens & Collins, 1977). Alternatively, the tutorial system might offer a menu of problems the student could be having, and allow the student to identify the relevant one. A simple system might have a predefined menu that was meant to cover all the problems a student might have, while a more advanced system might infer a smaller set of possible problems from the tutorial context. If a student working on the TWOS problem entered "FOR I = 2 TO 8" and requested a hint, the system might offer the menu:

```
Do you want help on:
        1: the STEP component of the FOR statement
        2: the NEXT statement
        3: the use of variables in FOR statements
        4: something else
        5: nothing (you don't want any help)
```

These topics for possible hints would be identified by comparing the correct statement to the statement entered by the student and offering help about each of the mismatches between the statements (e.g., the absence of the STEP component in the FOR statement, the unentered NEXT statement, and the improper use of a numerical constant as the upper bound of the loop) or some other problem not included in this list, as well as the option of rejecting help.

3.5 Advantages of a Human Tutor

What are the properties of a human tutor that make his suggestions more valuable to a student than the help offered by BIP? The assistance available from the tutor and from BIP differed in two ways:

* Flexibility: Unlike BIP's help messages, which are tied to specific command or statement errors, a human tutor can, at any time, offer help on any part of the tutorial interaction. In addition, a tutor can properly interpret a very large number of ways of solving a particular problem, whereas BIP's diagnosis of a student's program assumes a particular approach to the problem.

* Specificity: Although BIP can describe in general terms the statements that comprise a program (i.e., "... you need an IF statement and two PRINT statements..."), identify illegal statements, and offer assistance on the proper forms of these statements when errors occur, it can offer very little help once the student has entered a statement that is syntactically legal, but incorrect in the context of the current problem. In contrast, a human tutor can evaluate a legal statement and determine that some subpart of the statement -- perhaps the variable that specifies the upper bound of a FOR statement -- is at fault. Recall that the original version of BIP does have a "hint" facility, through which students can receive hints by issuing the BIP command "HINT", but since these hints are predefined in much the same way as the help BIP offers after statement errors, this facility can offer neither flexibility nor specificity. For instance, BIP's hint for TWOS is:

> The -FOR- statement can make the loop count by twos automatically. Look for an explanation of the 'step' part of the statement.

Although this hint may sometimes be useful, it would not have been much help to DD, who knew that she had to specify STEP information, but did not know the right way to do this.

In human tutors these properties come from knowledge of the problem at hand, general programming and problem solving techniques, and a variety of educational and tutorial strategies. The second portion of this report describes a system that, through its partial representation of these kinds of knowledge, can give a student more flexible and specific tutorial assistance. While this work is only a beginning, it brings us a step closer to more "intelligent" computer-based tutoring.

Section 4

Program Understanding and Synthesis in LOGO: The TURTLE Tutor

TURTLE is a tutorial system that provides "intelligent" assistance for a student solving a set of tasks in LOGO programming: this assistance is intended to be appropriate to the student's plan for solving the task and specific to the problem facing the student when help is requested. TURTLE uses analysis by synthesis techniques to interpret the student's program, to generate hints upon request by the student, and to assist the user in correcting flawed programs.

LOGO programs draw figures on the screen of a graphics terminal by simulating the movement of a "turtle". The turtle can draw a line on the screen corresponding to its movement. At the beginning of a session, the turtle is in the center of the screen, facing north; its position can be changed with the following primitives[8]:

-DRAW X: move forward X units, drawing a line.

-MOVE X: move forward X units without drawing a line.

-TURN X: turn the turtle X degrees clockwise.

A student's program can be made up of these primitives and references to user defined functions. The current implementation of TURTLE does not support recursion, iteration, variable assignment, or subroutine calls more than one level deep.

---

[8] Note that these primitives differ slightly from those described by Papert (1980), and in fact correspond to those defined in OGOL (Section 2).

Programs that meet these constraints can draw simple pictures like TRIANGLE, TREE, WELL, and NAPOLEON (Figure 2), the four tasks that form the current version of TURTLE.

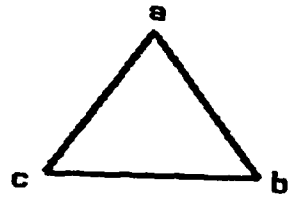
A student working with TURTLE is given the task of writing a LOGO program that will draw one of these figures. This program may simply be a series of LOGO statements -- an open coded program -- or it may decompose the figure into a number of subfigures, each of which is drawn by a function defined by the student. For instance, the WELL (see Figure 2) can be decomposed into the triangle at the top of the figure, the square at the bottom, and the line that connects the triangle and the square.[9] In whatever way the program is written, TURTLE's task is to identify the plan used by the student to construct this program and to use this plan to help the student locate and correct errors in the program.

4.1 Sample TURTLE Sessions for the TRIANGLE and WELL Tasks.

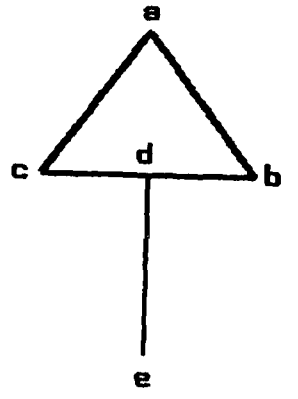
When a student begins a session with TURTLE, he is presented with a general discussion of computers and the LOGO language. During this discussion, he can experiment with the various LOGO primitives by entering statements and observing their results. The student then solves the TRIANGLE and TREE tasks. Next, the concept of program decomposition is introduced, and the student is encouraged to use decomposition to solve WELL and NAPOLEON.

---

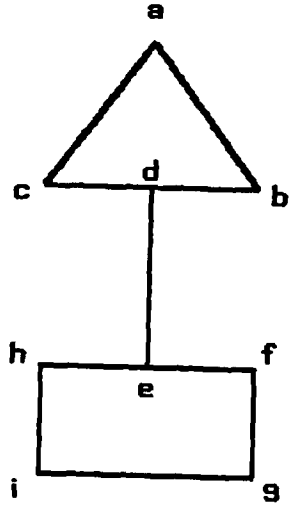
[9] Other decompositions are possible, of course; TURTLE's analysis of problems with several different decompositions is described in Section 4.2.5.



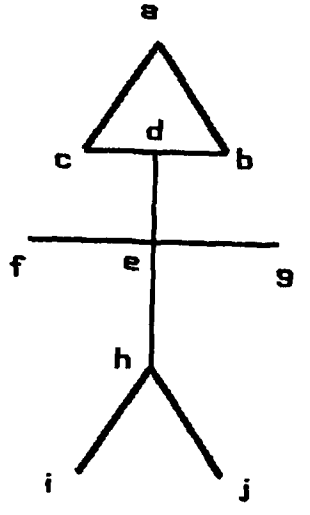
TRIANGLE



TREE



WELL



NAPOLEON

Figure 2 Sample TRIANGLE, TREE, WELL, and NAPOLEON Figures

4.1.1 TRIANGLE: Interpreting Open Coded Solutions

The first task presented to students by TURTLE was TRIANGLE, in which the student was to write an open coded program to draw an equilateral triangle (see Figure 2; note that the triangle is "pointing" upward). The following is a trace of a hypothetical subject's solution of this problem; TURTLE's output is in boldface. This section presents only a sample interaction with TURTLE -- a detailed description of how TURTLE operates will follow.

TURTLE first displays the correct solution of the triangle task on the graphics terminal; this figure can be drawn by a program that starts at vertex A (Figure 2) and proceeds as follows:

```
TURN 30
DRAW 200
TURN 120
DRAW 200
TURN 120
DRAW 200
```

The student is then asked to define TRIANGLE:

```
TRIANGLE is not yet defined.  Please enter the steps in
TRIANGLE.  (Type QUIT to end the function definition,
RESTART to start over, and DELETE to delete the last
step.)

First step? DRAW 200
Next step?  TURN 60
Next step?  DRAW 200
Next step?  QUIT
OK, TRIANGLE is defined!
```

There are several errors in the student's proposed program. Only two of the three sides of the triangle are drawn, and the

rotation between the first two steps should be TURN 120, rather than TURN 60. Despite the presence of these errors, TURTLE discovers three paths through the triangle that provide relatively good matches to the student's program, and asks the student which of these corresponds to his path through the figure:

```
Did you to intend to follow path C A B C? NO
Did you to intend to follow path B C A B? NO
Did you to intend to follow path A B C A? YES
```

Having identified the path used by the student, TURTLE generates the following analysis of the student's program, which, if desired, can be presented to the student:

Analysis of your program:

Student Code	Correct Code	Commentary
DRAW 200	DRAW 200	Correct mainstep to draw line segment AB
TURN 60	TURN 120	Incorrect interface to turn through angle ABC
DRAW 200	DRAW 200	Correct mainstep to draw line segment BC
	TURN 120	Missing interface to turn through angle BCA
	DRAW 200	Missing mainstep to draw line segment CA

TURTLE then displays the figures corresponding to desired solution and the student's current solution on the graphics terminal, and begins the debugging process:

Disregarding the orientation for now, what bugs, if any, do you see in the function TRIANGLE?

1 - NO BUGS
2 - MISSING STEPS
3 - INCORRECT STEPS
4 - EXTRA STEPS

> 2

The student says that there are missing steps; TURTLE agrees, and offers further information:

Yes, there are MISSING STEPS in the function TRIANGLE. You did not mention the following bugs: The function TRIANGLE has INCORRECT STEPS.

At this point, TURTLE shows the student his program and allows him to make modifications or to request a hint; the student requests a hint.

Here are the steps you are editing:

DRAW 200
TURN 60
DRAW 200

What would you like to do?

1 - SHOW FUNCTION
2 - BUGS REPORT
3 - DELETE STEP
4 - CHANGE STEP
5 - INSERT STEP
6 - RECEIVE HINT
7 - FINISH EDITING

> 6
[RECEIVE HINT]

Here, TURTLE offers a general hint about the missing step -- You are missing the line segment from C to A -- after which the student asks for another hint. This hint is more specific:

You are missing the step DRAW 200 to draw the line segment from C to A.

At this point, the student has been returned to the editing menu, asks to insert a step, and enters DRAW 200 at the end of his program, which will draw the third side of the triangle. He then requests another hint, and receives help on one of the missing TURNs:

You are missing the rotation to turn through angle BCA.

When this hint is displayed, the line segments that make up angle BCA are highlighted in the displays of the correct drawing and of the results of the student's current program. The student then asks to enter the step TURN 60 between the latter two DRAW 200's. However, TURTLE rejects this addition:

That step does not need to be inserted anywhere. A TURN does need to be inserted, but the value 60 is incorrect.

At this point, the student determines that a 120 degree turn is needed, and enters a TURN 120; the result of this revised program is then displayed. The student now asks for another hint; TURTLE replies

The rotation to turn through angle ABC is incorrect.

and highlights the rotations in both the correct drawing and the student's current drawing. The student continues by requesting another hint, and is told:

The rotation to turn through angle ABC should be TURN 120 instead of TURN 60.

Here, the student uses the CHANGE STEP option in the editing menu to change the TURN to 120 degrees, and then selects FINISH

EDITING. The function is now correct, and TURTLE reports: The **function TRIANGLE has no bugs!**

Although the function itself is correct, the figure's orientation is not: the triangle "points" to the right, rather than upward. TURTLE now asks the student to specify interface steps that will correct this orientation error:

> **You provided no setup for the TRIANGLE-TASK. What bugs, if any, do you see in the orientation of TRIANGLE?**

The student says that there are missing steps, and TURTLE agrees:

> **Yes, there are MISSING STEPS in the orientation of TRIANGLE. No setup interface for TRIANGLE has been provided. What would you like to do?**

Here, the student asks for a hint regarding the interface, and, after receiving it --

> **Another TURN command is necessary to orient TRIANGLE correctly.**

inserts it at the correct place. The student again asks to FINISH EDITING, and TURTLE tells him that he has completed the task.

4.1.2 WELL: Decomposing a Complex Figure

The following section describes TURTLE's tutoring of the more complex WELL task, and focuses on how TURTLE recognizes standard task decompositions and helps the student debug interfaces and figure orientation.

The session begins with the display of a wishing well (see Figure 2); TURTLE then asks the student to enter his program, here called WELL.

```
First step? TOP
Next step?  DRAW 200
Next step?  BOX
Next step?  QUIT
OK, WELL is defined!
```

The student has decomposed the WELL into three steps: his plan is to draw the top of the WELL with the function TOP, the base with the function BOX, and the POLE with the open coded DRAW 200. TURTLE now determines the student's plan from this program:

```
Let me see if I understand your program:
The function TOP draws the ROOF of the WELL-TASK.
The code: DRAW 200 draws the POLE of the WELL-TASK.
The function BOX draws the BASE of the WELL-TASK.

Does that seem correct? YES
```

By identifying the student's plan for the program -- in particular, the purposes of the TOP and BOX functions and the open coded DRAW -- TURTLE will be able to identify errors and offer the same kinds of advice that it did in the TRIANGLE example. TURTLE now asks the student to define either TOP and BOX; the student chooses to work on TOP first.

```
TOP is not yet defined.  Please enter the
steps in TOP.

First step? DRAW 100
Next step?  TURN 120
Next step?  DRAW 100
Next step?  TURN 120
Next step?  DRAW 100
Next step?  TURN 180
Next step?  QUIT
OK, TOP is defined!
```

Note that this is identical to the definition of TRIANGLE shown previously, except for the addition of TURN 180 at the end of the program, which is meant to put the the turtle in the proper position for drawing the WELL's POLE. (This statement alone is not sufficient; this error will be corrected later.) TURTLE now tries to determine the plan underlying the student's TOP program, and again asks which of the three possible paths through the WELL's triangle corresponds to the student's plan:

```
Did you to intend to follow the path B C A B? NO
Did you to intend to follow the path A B C A? NO
Did you to intend to follow the path C A B C? YES
```

Using this plan, TURTLE determines that the student's function is correct; the final TURN 180 is classified as part of the interface between the TOP and the (open coded) POLE. TURTLE now moves the student on to the definition of the BOX function:

```
BOX is not yet defined.  Please enter the steps in BOX.
First step? DRAW 100
Next step?  TURN 90
Next step?  DRAW 100
Next step?  TURN 90
Next step?  DRAW 100
Next step?  TURN 90
Next step?  DRAW 100
Next step?  QUIT
OK, BOX is defined!
```

As before, TURTLE determines the path the student planned to take through the box:

```
Did you to intend to follow the path F G I H F? YES
```

and notes that the code for BOX is correct.

Now that the student has defined the three components of the figure, he must define a series of interface steps that will properly position these parts of the program with respect to each

other. TURTLE first checks its hypothesis about the interface between the TOP and the open coded POLE:

> Did you intend to interface the function TOP to the step DRAW 200 by following the path C D ? YES

TURTLE now shows the student whatever interface steps are already present in his program (here, there is only one -- TURN 180) and compares them to the correct interface:

> Here is the interface you defined between the mainsteps in the function TOP and the step DRAW 200:
>
> TURN 180
>
> Analysis of your interface between the function TOP and the step DRAW 200 in WELL:

Student Code	Correct Code	Commentary
TURN 180	TURN 180	Correct interface to turn through angle BCD
	MOVE 50	Missing interface to move over line segment CD
	TURN 90	Missing interface to turn through angle CDE

TURTLE now guides the student through these modifications until the correct interface has been constructed. Similar steps are taken in constructing an appropriate interface between the open coded POLE and the BOX. TURTLE then checks the orientation of the entire figure and detects another error: although the components of the figure itself are correct, the figure as a whole is at an incorrect angle. TURTLE asks the student about errors in this orientation --

> You provided no setup for the WELL-TASK.
> What bugs, if any, do you see in the orientation of WELL?

Here, the student requests and receives a hint -- **Another TURN command is necessary to orient WELL correctly** -- and enters the necessary step -- TURN 30. The figure is now constructed correctly, and the session ends.

4.2 TURTLE: Plan Understanding via Analysis by Synthesis

The key to any successful tutorial system is to understand the plan that the student has generated to solve the problem at hand. In this project, this key is understanding the design of the student's LOGO program and the function of each of the steps of the program. To this end, TURTLE exploits the constraints of the turtle graphics domain and the restricted subset of the LOGO language used here: TURTLE's first step in identifying the design of a student's program is to generate the entire set of "viable"[10] paths through the figure and to compare them to the path drawn by the student's program. Since plans can be associated with these candidate paths when they are generated, the problem of identifying the student's plan is reduced to the problem of finding the path that offers the best match to the student's program.

This process requires three classes of knowledge structures; the remainder of this section describes these structures and how they are used to achieve the goals of appropriate and specific student assistance. TURTLE's curriculum structure specifies the

---

[10] Viable paths are those with a limited number of path retraces (repeated traversals of the same line segment) -- typically no more than one -- and with line segments that are broken no more than once.

textual information and function calls that comprise the tutorial session. Its task networks provide thorough descriptions of the figures to be drawn by students and are essential to TURTLE's analysis by synthesis evaluation of a student's program. TURTLE uses a figure's task network to synthesize a number of programs capable of drawing this figure. Differences between these programs and the student's proposed program are then noted by means of annotations to the synthesized programs. These annotations are used to identify the student's plan, by finding the synthesized program that most closely matches the student's program, and to guide the construction of hints, by means of a set of hint generation structures that detect particular error annotations and construct hints that reflect the presence of these errors.

4.2.1 Curriculum Structure

For each problem in the curriculum that is to be presented by TURTLE, two functions are defined: a presentation function that will carry out the actual tutoring of the problem, and a successor function that will determine the next problem that should be presented to the student. An executive function then retrieves and executes the presentation function for the selected problem (thereby carrying out this problem's tutorial session), and, by executing the successor function, selects the next problem that will be given to the student. The successor function can, in principle, base this decision on such factors as the student's performance on the current and previous problems. However, at this early stage in TURTLE's development, the problems' successor functions simply pass the student through the

TRIANGLE, TREE, WELL, and NAPOLEON tasks, in that order.

4.2.2 Task Representation

As described previously, TURTLE generates all viable paths through a figure and then matches the student's program against these paths. To generate these paths, TURTLE must have a representation of the figure that is explicit enough to support path generation, yet general enough to allow for errors in the student's program and for variations in the size and orientation of the figure drawn by the student.

TURTLE uses a system of task networks to organize this information. The task networks for the TRIANGLE and WELL figures are shown in Tables 7 and 8.

These networks are frame-like units with the following slots:

-**TASK-NAME**: the name of the task.

-**FIGURE-GEOMETRY**: a specification of the vectors and angles that comprise the figure. This slot specifies:

  -**VERTICES**: the points of the figure that are to be connected.

  -**CONNECTIONS**: the line segments that connect the figure's vertices. This slot also contains information about the size and possible subdivision of the line segments.

  -**INTERFACES**: the size of angles formed by connected line segments.

  -**STARTING-POINTS**: the vertices that may be used to begin the figure's construction.

-**DESIRED-ORIENTATION**: the orientation of a particular line segment, by which the orientation of the entire figure can be determined.

-**POSSIBLE-NAMES**: a list of possible names for the figure and any subfigures.

Table 7:

**TURTLE's task network for TRIANGLE.
References to vertices correspond to those of the figures shown in Figure 2.**

```
TASK-NAME: TRIANGLE

FIGURE-GEOMETRY:

        VERTICES:
                (a b c)

        CONNECTIONS:
                ((join a b)
                        via (line-segment 1 x))
                ((join a c)
                        via (line-segment 1 x))
                ((join b c)
                        via (line-segment 1 x))
                (default-for x 100)

        INTERFACES:
                (c a b angle 60)
                (b c a angle 60)
                (a b c angle 60)

        STARTING-POINTS:
                (start-at (a b c))

DESIRED-ORIENTATION:
        (orientation a b should-be 150)

EXPECTED-DECOMPOSITIONS:
                (triangle -> open-code))

EXAMPLE-SOLUTION:

        ((turn 30)
         (draw 1 x)
         (turn 120)
         (draw 1 x)
         (turn 120)
         (draw 1 x))
        ((setup-for-task)
         (draw-line-segment c a)
         (turn-through-angle c a b)
         (draw-line-segment a b)
         (turn-through-angle a b c)
         (draw-line-segment b c))
```

Table 8:

TURTLE's task network for WELL.

```
TASK-NAME: WELL

FIGURE-GEOMETRY:

     VERTICES:
          (a b c d e f g h i)

     CONNECTIONS:
          ((join a b)                          ;Roof of the well
               via (line-segment 2 x))
          ((join a c)
               via (line-segment 2 x))
          ((join b c)
               via (line-segment 2 x)
               contains ((b d) (c d))
               subdivided-by d)
          ((join b d)
               via (line-segment 1 x)
               contained-in b c)
          ((join d c)
               via (line-segment 1 x)
               contained-in b c)
                                               ;Pole of the well
           ((join d e)
               via (line-segment 1 z))
                                               ;Base of the well
           ((join h f)
               via (line-segment 2 y)
               subdivided-by e
               contains ((h e) (e f)))
           ((join f g)
               via (line-segment 2 y))
           ((join g i)
               via (line-segment 2 y))
           ((join i h)
               via (line-segment 2 y))
           ((join h e)
               via (line-segment 1 y)
               contained-in h f)
           ((join e f)
               via (line-segment 1 y)
               contained-in h f)
           (default-for (x 100) (y 100) (z 200))

        INTERFACES:
               (d c a angle 60)   (a b d angle 60)
               (c a b angle 60)   (b c a angle 60)
               (a b c angle 60)   (e f g angle 90)
               (i h e angle 90)   (g i h angle 90)
               (f g i angle 90)   (h f g angle 90)
```

```
                (i h f angle 90)   (d e h angle 90)
                (f e d angle 90)   (e d b angle 90)
                (c d e angle 90)   (c d b angle 180)
                (b c d angle 0)    (c b d angle 0)
                (h e f angle 180)  (f h e angle 0)
                (h f e angle 0)

        STARTING-POINTS:
                (a b c d e f g h i))

DESIRED-ORIENTATION:
        (orientation i g should-be 90)

POSSIBLE-NAMES:
        (roof is-also-called (triangle roof ro tri tr top))
        (pole is-also-called (pole po line li line-segment
                                    vect middle mid))
        (base is-also-called (base ba square sq squ bottom
                                    bot bo))
        (tree is-also-called (tree))

EXPECTED-DECOMPOSITIONS:
        (well -> roof pole base
                (Vertices:
                        ((roof a b c d)
                         (pole d e)
                         (base h e f g i)))
                (Termination-Points:
                        (((roof ending-point) d)
                         ((pole starting-point) d)
                         ((pole ending-point) e)
                         ((base starting-point) e))))
        (well -> base pole roof
                (Vertices:
                        ((roof a b c d)
                         (pole d e)
                         (base h e f g i)))
                (Termination-Points:
                        (((roof starting-point) d)
                         ((pole ending-point) d)
                         ((pole starting-point) e)
                         ((base ending-point) e))))
        (well -> tree base
                (Vertices:
                        ((tree a b c d e)
                         (base h e f g i)))
                (Termination-Points:
                        (((tree ending-point) e)
                         ((base starting-point) e))))
        (well -> base tree
                (Vertices:
                        ((tree a b c d e)
                         (base h e f g i)))
                (Termination-Points:
                        (((tree starting-point) e)
                         ((base ending-point) e))))
```

```
        (well -> open-code)

EXAMPLE-SOLUTION:
        ((turn 270)
         (draw 1 x)
         (turn 120)
         (draw 2 x)
         (turn 120)
         (draw 2 x)
         (turn 120)
         (draw 1 x)
         (turn 270)
         (draw 1 z)
         (turn 90)
         (draw 1 y)
         (turn 270)
         (draw 2 y)
         (turn 270)
         (draw 2 y)
         (turn 270)
         (draw 2 y)
         (turn 270)
         (draw 1 y))
        ((setup-for-task)
         (draw-line-segment d c)
         (turn-through-angle d c a)
         (draw-line-segment c a)
         (turn-through-angle c a b)
         (draw-line-segment a b)
         (turn-through-angle a b d)
         (draw-line-segment b d)
         (turn-through-angle b d e)
         (draw-line-segment d e)
         (turn-through-angle d e h)
         (draw-line-segment e h)
         (turn-through-angle e h i)
         (draw-line-segment h i)
         (turn-through-angle h i g)
         (draw-line-segment i g)
         (turn-through-angle i g f)
         (draw-line-segment g f)
         (turn-through-angle g f i)
         (draw-line-segment f e))))

        (default-bindings well ((X 100.0) (Y 100.0) (Z 200.0)))
```

-**EXPECTED-DECOMPOSITIONS**: descriptions of how the figure might be decomposed. For each of the figures that can be decomposed (TRIANGLE cannot), this slot lists the sets of vertices that comprise these subfigures and the acceptable beginning and ending points for the subfigures.

-**EXAMPLE-SOLUTION**: a LOGO program that, once exact values for the line segments have been determined, will draw the figure correctly. This structure also contains general descriptions of the function of each of the statements of the program, such as DRAW-LINE-SEGMENT and TURN-THROUGH-ANGLE.

4.2.3 Program Synthesis

These task networks provide the descriptive information TURTLE uses to generate the viable paths through a figure and the LOGO programs that correspond to these paths. The resulting paths can then be matched against the student's code, errors can be detected and diagnosed, and appropriate hints can be given.

This path generation depends upon the specification of the figure's STARTING-POINTs and CONNECTIONS. TURTLE generates all viable paths from each of the figure's starting points, as determined by the connections between the figure's vertices. In TRIANGLE's task-network, since all three of the triangle's vertices are specified as starting points, paths through the triangle would be generated from vertices A, B, and C. The connection information would then guide the path construction process: two paths would be built from vertex A, based on the line segments between vertices A and B and between A and C. These paths are then expanded in a similar way: for instance, the "A -> B" path is expanded to "A -> B -> C" via the line segment from B to C. This generation process operates under three constraints. First, all the figure's line segments must appear in the final

path. Second, a line segment can be traversed more than once, but only one of the traversals can be a DRAW; the rest must be retraces that MOVE over the already drawn line. Third, no more than a fixed number of retraces may take place -- if retraces were not limited, an infinite number of open coded solutions would exist for even the simplest figures. Our experiments with TURTLE limited the number of acceptable retraces to one.

This technique produces six possible paths for the TRIANGLE figure:

A -> B -> C -> A	A -> C -> B -> A
B -> C -> A -> B	B -> A -> C -> B
C -> A -> B -> C	C -> B -> A -> C

These paths are then converted into LOGO code by referring to the task network's specifications of line segments and interface angles. Consider the path "A -> B -> C -> A". The first step in this path, "A -> B", corresponds to the CONNECTIONS entry:

((JOIN A B) VIA (LINE-SEGMENT 1 X))

(see Table 7) and a DRAW is generated. Since the absolute size of the figure that will be drawn by the student cannot be predicted, the length of this line segment is specified in terms of a constant and a variable. The "1 X" in the "(LINE-SEGMENT 1 X)" structure indicates the desired length of the line segment as the product of the scale factor, 1, and a variable, X; the pseudo-LOGO statement "DRAW 1 X" is generated. These constant-variable pairs allow TURTLE to interpret student programs of any absolute size; the only requirement for a correct program is that the relative sizes of the figure correspond to

those defined in the task network. Default values for these variables are specified so that TURTLE can generate executable LOGO code from these structures before explicit line segment lengths have been specified.

The next step in this path is "B -> C"; here, TURTLE should build the code not only for the DRAW from B to C, but also the TURN that will draw this line segment in the proper direction. This TURN through the angle ABC is derived from the angle's INTERFACE specification:

(A B C ANGLE 60)

The instruction "TURN 120" results: the 60 degree interior angle of the triangle is generated by turning the turtle through the 120 degree exterior angle. The remaining instructions for this path and for the other five paths are generated in this way, producing six solution structures:

```
(SOLUTION1              (SOLUTION2              (SOLUTION3
        (A B C A)               (B C A B)               (C B A C)
        (DRAW 1 X)              (DRAW 1 X)              (DRAW 1 X)
        (TURN 120)              (TURN 120)              (TURN 120)
        (DRAW 1 X)              (DRAW 1 X)              (DRAW 1 X)
        (TURN 120)              (TURN 120)              (TURN 120)
        (DRAW 1 X))             (DRAW 1 X))             (DRAW 1 X))

(SOLUTION4              (SOLUTION5              (SOLUTION6
        (A C B A)               (C B A C)               (B A C B)
        (DRAW 1 X)              (DRAW 1 X)              (DRAW 1 X)
        (TURN 240)              (TURN 240)              (TURN 240)
        (DRAW 1 X)              (DRAW 1 X)              (DRAW 1 X)
        (TURN 240)              (TURN 240)              (TURN 240)
        (DRAW 1 X))             (DRAW 1 X))             (DRAW 1 X))
```

4.2.4 Program Recognition

Once these solution structures have been generated, a student's proposed program can be matched against these

structures. Since an exact match will occur only if the program is correct, partial matching techniques are used to permit and identify differences between the proposed and correct solutions.

Recall the illustrated TRIANGLE task once again: the student's proposed program was:

```
DRAW 200
TURN 60
DRAW 200
```

This program must be matched against the six proposed solutions shown above. The first step in this process is to build a matching structure from this program; this is done by replacing each argument of the program's TURN statements with position variables, which can match any angle value -- including the value of incorrect turns -- and by inserting before and after each program statement segment variables, which can match any number of statements and so can identify statements the student has incorrectly entered or omitted from his program. This procedure produces the structure:

$S0 (DRAW 200) $S1 (TURN ?ANGLE) $S2 (DRAW 200) $S3)

which must be matched against the proposed solutions, perhaps SOLUTION1:

(DRAW 1 X) (TURN 120) (DRAW 1 X) (TURN 120) (DRAW 1 X)

This match will succeed, as the result of four separate steps. First, the program's first DRAW matches the solution's first DRAW, giving X -- the variable used to specify the lengths of line segments -- a value of 200. Note that the matcher attempts to match DRAWs wherever possible, rather than letting

DRAWs be matched by segment variables. As a result, the segment variable $S0 receives no value. Second, the program's TURN matches the solution's TURN, with the position variable ?VALUE set to 120 (not 60, as in the student's original program). Since this successfully matched TURN immediately followed the previously matched DRAW, $S1 receives no value. Third, the program's second DRAW matches the solution's second DRAW; the success of this match results in $S2 receiving no value. Fourth, since the student's program ends with this second DRAW, the segment variable $S3 is matched to the remaining statements in the solution -- (TURN 120) and (DRAW 1 X).

This technique is used to match the student's program against the proposed solutions. Each of these matches is then given penalty points that determine the match's "badness of fit"; the following ad hoc rules are used to award these points:

- -Missing steps: 1 point.
- -Incorrect TURNs:
  - -If the sum of the angles is 180 degrees, .25 point. This rule will trap the common error of specifying turns by their interior, and not exterior, angles.
  - -If the proposed and correct turns are both clockwise (less than 180 degrees) or both counterclockwise (between 180 and 360 degrees), .5 point.
  - -Otherwise, 1 point.

This method awards SOLUTION1 above a score of 2.5 points: 2 points for the two missing steps, and .5 point for the TURN statement: the exact angle of the TURN is wrong, but, like the corresponding statement in the solution, it is in a clockwise

direction. Since SOLUTION1, SOLUTION2, and SOLUTION3 all use clockwise turns, they all will receive scores of 2.5. Similarly, since SOLUTION4, SOLUTION 5, and SOLUTION6 are identical, but use counterclockwise turns to solve the problem, they will receive scores of 3 points. The incorrect TURN in these solutions receives a penalty score of 1 point.

SOLUTION1, SOLUTION2, and SOLUTION3 are retained as candidates for the correct interpretation of the student's plan, since they share the score that indicated the best program-solution match. However, TURTLE cannot determine which of these three solutions is correct without knowing which of the vertices of the triangle was used by the student as the starting point of his program. This problem can be resolved only by asking the student which of the three hypothesized paths corresponds to his own:

```
Did you to intend to follow path C A B C? NO
Did you to intend to follow path B C A B? NO
Did you to intend to follow path A B C A? YES
```

In this way, the correct solution -- here, SOLUTION1 -- can be identified, and the plan corresponding to this solution can be used to analyze and correct the errors in this program.

4.2.5 Program Decomposition

The simplicity of the TRIANGLE task makes the identification of the student's plan very straightforward. However, the technique described above will be less successful as the tasks to which it is applied become more complex and the number of possible paths through these figures increases. This problem can

be avoided by encouraging the student to decompose a large problem into a set of smaller problems. Problem decomposition is typically good programming style and would simplify TURTLE's task considerably. It would have to do the kinds of analysis shown above only on simple figures such as triangles and squares, where the number of viable paths is small. However, this simplification comes at a price: TURTLE must be able to identify how the student is decomposing the problem, so that proper interpretations can be given to the program segments that carry out the individual tasks.

TURTLE interprets a student's decomposition in much the same way it determines the plan underlying an open coded program. Its strategy is based on two assumptions about how people will solve these problems:

- People will decompose a problem that draws a complex figure into a set of functions that draw regular figures such as triangles, rectangles, and circles.
- People will use mnemonic names for the functions that carry out particular subparts of the problem: a function that draws the square base of the well will probably be called BASE or SQUARE.

TURTLE exploits these assumptions in the following ways. The first assumption implies that TURTLE need anticipate only those decompositions made up of regular figures. Descriptions of these probable decompositions can then be included in a figure's task network; the expected decompositions for WELL include structures such as the following (the full set of expected decompositions can be found in WELL's task network in Figure 8):

```
EXPECTED-DECOMPOSITIONS:
        (WELL -> ROOF POLE BASE
                 (VERTICES:
                         ((ROOF A B C D)
                          (POLE D E)
                          (BASE H E F G I)))
                 (TERMINATION-POINTS:
                         (((ROOF ENDING-POINT) D)
                          ((POLE STARTING-POINT) D)
                          ((POLE ENDING-POINT) E)
                          ((BASE STARTING-POINT) E))))
```

This structure states that the WELL consists of a ROOF, a POLE, and a BASE. The ROOF is drawn by vertices A, B, C, and D (see Figure 2), the POLE by vertices D and E, and the BASE by vertices E, F, G, H, and I. The WELL can also be drawn as a TREE and a BASE; a structure that represents this decomposition is also included in WELL's task network. TURTLE will use some subset of these four names to internally describe how the student has decomposed the WELL problem. All programs involving decomposition will be determined to be some collection of the subfigures ROOF, POLE, BASE, and TREE.

The identification of students' decompositions would be simple if students used only these four names for the subfigure functions. This, of course, is not the case. However, since people will probably give the subfigure functions mnemonic names, the number of probable function names is limited. Like the figure decompositions, these names can be enumerated and included in the figure's task network; the names recognized by TURTLE for the functions used to draw the possible subparts of WELL -- ROOF, POLE, TREE, and BASE -- are structured as follows:

```
POSSIBLE-NAMES:
        (ROOF IS-ALSO-CALLED (ROOF RO TRIANGLE TRI TR
                                   TOP))
        (POLE IS-ALSO-CALLED (POLE PO LINE LI
                                   LINE-SEGMENT VECT
                                   MIDDLE MID))
        (BASE IS-ALSO-CALLED (BASE BA SQUARE SQ SQU
                                    BOTTOM BOT BO))
        (TREE IS-ALSO-CALLED (TREE))
```

These structures are then used to identify the decomposition used by a student. Recall the proposed program for the WELL task (Section 4.1.2):

```
TOP
DRAW 200
BOX
```

The decomposition represented by this program is determined by converting the program into a matching structure and comparing it to the possible decompositions. The construction of this matching process requires three steps:

- Each student function name that appears in the POSSIBLE-NAMES structure is replaced by the concept under which the function name is indexed. By this rule, the student function name TOP is converted to ROOF.

- Each function name that does not appear in the POSSIBLE-NAMES structure is converted to a position variable: since BOX is not a name known to TURTLE, it is replaced by the position variable ?BOX.

- All strings of LOGO function calls are replaced by segment variables: the DRAW 200 statement is replaced by $S0.

This conversion process produces the matching structure

(ROOF $S0 ?BOX)

which can be matched against the set of possible decompositions of the WELL; as before, $S0 can match any string of statements, and ?BOX can match any single item. The best match comes from

the decomposition

(ROOF POLE BASE)

which leads to TURTLE's interpretation of the student's program;

```
Let me see if I understand how you are going to solve
the WELL-TASK:
The function TOP draws the ROOF of the WELL-TASK.
The code: DRAW 200 draws the POLE of the WELL-TASK.
The function BOX draws the BASE of WELL-TASK.

Does that seem correct? YES
```

Note that the matching process will fail if the student writes a program with function names that are recognized by TURTLE, but that appear in an unexpected order, such as ROOF -> BASE -> POLE. When given such a program, TURTLE will ask the student to define the paths to be drawn by each of these programs; if these completely describe the figure to be drawn, it will proceed with this decomposition. *TURTLE* also allows the student to define new decompositions through a similar technique of asking the student to define the paths taken by the new functions.

4.2.6 Specifying Setup Orientations and Interfigure Interfaces

The discussion thus far has concentrated on the generation of programs that correctly draw the basic figures that make up the tasks discussed here -- triangles, squares, and the like. There are two further tasks, however, that affect the correctness of a LOGO program. These are concerned with the spatial orientation of the program's components.

As noted in the sample TRIANGLE session, the following code generates a correct triangle:

```
DRAW 100
TURN 120
DRAW 100
TURN 120
DRAW 100
```

Since the student began this figure at vertex A, the above code will produce a triangle that "points" to the right. Since the turtle begins a session facing due north, the first DRAW also goes north. Facing north is only one of many possible orientations, however, and is in fact incorrect. As shown in Figure 2, the generated triangle must point upward. The program needs an additional "setup" step at the beginning of the program -- TURN 30 -- to place the triangle in its proper orientation.

The figure's orientation is specified in TRIANGLE's task network as its DESIRED-ORIENTATION:

(ORIENTATION A B SHOULD-BE 150)

This form states that the line segment from A to B should form a 150 degree angle with a horizontal line. Since the angles drawn in turtle-graphics systems are assumed to be rigid, the entire orientation of an otherwise correct figure can be determined by examining this one segment.

TURTLE determines the current orientation of this line segment by generating (via the figure's task network CONNECTIONS) a path from the line segment backward to the starting point of the figure, which the student identified when he verified TURTLE's interpretation of his plan. Since the line segment drawn at the very beginning of the figure will have an

orientation of zero degrees,[11] and the angles between the line segments that make up this path are also specified in the task network, the actual orientation of AB can be computed. TURTLE can then determine the size of the "setup" turn from the difference between the desired and actual orientations of AB; here, that difference is 30 degrees. The student can then be tutored on the construction of this interface.

The second orientation problem is caused by the decomposition of a program into simpler tasks. As shown earlier, the WELL problem can be logically subdivided into three subtasks -- a square, a line, and a triangle. Both of the functions and the open coded line segment written by the student in the sample session (Section 4.1.2) were individually correct; however, these functions, when called one after another, produce the incorrect drawing shown in Figure 3.

One error in this figure is that no setup orientation is present; a more significant problem, however, is that the figures are not interfaced properly: the POLE leaves the TOP and enters the BOX at the wrong places. Although the student whose session appears in Section 4.1.2 tried to define an interface between TOP and the DRAW -- via the TURN 180 at the end of TOP -- additional statements are needed between all three of these statements to insure that the three figures are properly positioned. Unfortunately, no one set of interface statements exists that

---

[11] unless the program began with a TURN statement, in which case the first line segment will have an orientation corresponding to the angle of the TURN.

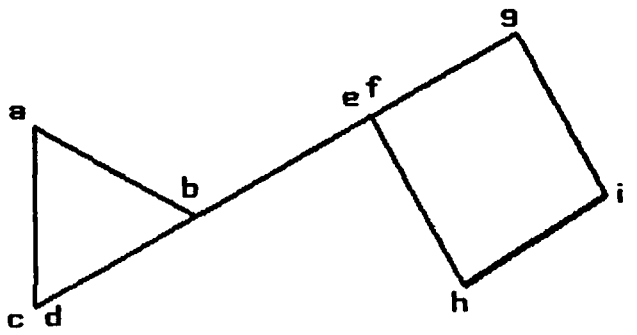


Figure 3 The WELL Figure Drawn by the Student's Initial (and Incorrect) Decomposition

might be identified and included in the figure's task network as were figure decompositions, because the student's design of the subfigures will determine the steps required to interface these figures. TURTLE must therefore wait until the student has defined the starting and ending points of his subfigures, and then generate interface steps that are appropriate for these functions.

When the interface between two subfigures is being generated, TURTLE has determined how the student has decomposed the task, and so knows which vertices are drawn by which subfigures. In the example above, TURTLE has determined that TOP follows the path C -> A -> B -> C, the DRAW that creates the WELL's POLE moves from D to E, and BOX follows the path F -> G -> I -> H. The interface steps between TOP and the DRAW must therefore move the turtle from C to D (from the ending point of TOP to the starting point of the DRAW) and position it properly for the subsequent DRAW. Similarly, the interface steps between the DRAW and BOX must move from E to F and position the turtle properly for the drawing of the BOX. These paths can be generated in much the same way as the paths that draw the individual figures: by generating all sequences steps through the figure, subject to the constraints on retraces described in Section 4.2.3. TURTLE generates the paths C -> D and C -> A -> B -> D as possible interfaces between vertices C and D. These paths are then converted to LOGO code, again in much the same way as the original programs -- by using the specifications of line segments and angles in the task's figure-geometry to convert the paths into an appropriate set of TURNs and DRAWs. Note that

TURTLE must refer to the steps immediately before and after the interface steps to determine the initial and final TURNs. To correctly turn onto the path from C to D, TURTLE must know the orientation of the turtle upon reaching vertex C, which would depend upon whether the student drew TOP along the path C -> A -> B -> C or the path C -> B -> A -> C. Since, in this example, TURTLE knows that the student followed the path C -> A -> B -> C in TOP, the following possible interface solutions result:

```
(SOLUTION1
        (C D)
        (TURN 180)   ; turn through angle DCD
        (MOVE 1 X)   ; move over line segment CD
        (TURN 90))   ; turn through angle CDE

(SOLUTION2
        (C A B D)
        (TURN 120)   ; turn through angle ACD
        (MOVE 2 X)   ; move over line segment AC
        (TURN 120)   ; turn through angle CAB
        (MOVE 2 X)   ; move over line segment AB
        (TURN 120)   ; turn through angle ABD
        (MOVE 1 X)   ; move over line segment BD
        (TURN 270))  ; turn through angle BDE
```

The student's interface is matched against these two possible solutions. As before, points are awarded that evaluate the quality of the match (although a different system for awarding point values is used here), and the solution with the better (i.e., lower) score is selected. Here, SOLUTION1 is the better solution -- its TURN 180 matches the student's interface exactly, and its number of steps is much closer to that of the student's plan -- and so will be used to debug the student's interface.

4.2.7 Program Annotation and Hint Generation

Once the plans underlying the student's programs and interfaces have been identified, TURTLE tries to identify any errors that may be present in the program. This identification is a two-step process. First, patterns are constructed that describe the intent of each statement of the program and whether that statement achieves that intent; second, these patterns are matched against predefined structures that reveal the presence of certain kinds of errors.

Program errors are interpreted by matching the student's actual program to the program that, as hypothesized by TURTLE, corresponds to the student's plan. The result of this match is a set of PROGRAM-STEP structures, one for each statement of the correct program. These structures are of the form:

```
(PROGRAM-STEP number
              step-representation
              evaluation
              intended-action)
```

A program-step's NUMBER is simply the ordinal number of the step in the correct program. The STEP-REPRESENTATION consists of an arbitrary and unique label for the step, followed by the correct statement itself. The EVALUATION states whether the step is CORRECT or INCORRECT in the student's program, or whether it is MISSING. The INTENDED-ACTION is a general description of the statement's function -- such as (DRAW-LINE-SEGMENT B C) -- which is obtained from the EXAMPLE-SOLUTION entry in the figure's task network.

This analysis can be demonstrated via the sample session

with the TRIANGLE task in Section 4.1.1. The proposed and correct programs for this task were:

Analysis of your program:

Student Code	Correct Code	Commentary
DRAW 200	DRAW 200	Correct mainstep to draw line segment AB
TURN 60	TURN 120	Incorrect interface to turn through angle ABC
DRAW 200	DRAW 200	Correct mainstep to draw line segment BC
	TURN 120	Missing interface to turn through angle BCA
	DRAW 200	Missing mainstep to draw line segment CA

The following program-step structures are built from these correct and incorrect programs:

```
(PROGRAM-STEP 1
              (T0014 DRAW 200)
              CORRECT
              (DRAW-LINE-SEGMENT A B))

(PROGRAM-STEP 2
              (T0015 TURN 60)
              INCORRECT
              (SHOULD-BE TURN 120)
              (TURN-THROUGH-ANGLE A B C))

(PROGRAM-STEP 3
              (T0016 DRAW 200)
              CORRECT
              (DRAW-LINE-SEGMENT B C))

(PROGRAM-STEP 4
              (T0020 TURN 120)
              MISSING-STEP
              (TURN-THROUGH-ANGLE B C A))

(PROGRAM-STEP 5
              (T0021 DRAW 200)
              MISSING-STEP
              (DRAW-LINE-SEGMENT C A))
```

In this analysis, program-steps 1 and 3 identify correct statements (the two DRAWs). Step 3 is incorrect (the turn should be 120 degrees), and steps 4 and 5 are missing.

When a student requests a hint, these pattern-step structures are matched against a set of hint generation structures (Table 9) that describe particular programming errors, such as the omission of a step or a TURN through an incorrect angle. Due to the constrained nature of the turtle-graphics task, the number of possible errors is finite and relatively small, and hint generation structures can be constructed for each. The hint generation structures are examined sequentially, and the first match found is used as the basis of a hint; different hinting strategies can therefore be established by ordering these structures in different ways.

TURTLE's hint generation rules are currently ordered so as to first detect and offer help on steps missing from the student's program. The first hint generated in the sample TRIANGLE task (Section 4.1.1) then focuses on the absence of the final DRAW in the student's program -- **You are missing the line segment from C to A.** This hint is the result of the match between the program-step structure corresponding to the missing DRAW:

```
(PROGRAM-STEP 5
              (T0021 DRAW 200)
              MISSING-STEP
              (DRAW-LINE-SEGMENT C A))
```

and the hint generation pattern:

Table 9:

TURTLE's hint generation structures, in the form:
(error-condition => general-hint specific-hint)

Hints for missing steps:

```
1: (program-step ? (? draw ?value) missing-step ?intention)

   =>

   ("You are missing the line segment from "
          (start-node ?intention) " to "
          (end-node ?intention) ".")

   ("You are missing the step DRAW " ?value
          " to draw the line segment from "
          (start-node ?intention) " to "
          (end-node ?intention) ".")


2: (program-step ? (? move ?value) missing-step ?intention)

   =>

   ("You are missing a retrace over the line segment from "
          (start-node ?intention) " to "
          (end-node ?intention) ".")

   ("You are missing the step MOVE " ?value
          " to retrace over the line segment from "
          (start-node ?intention) " to "
          (end-node ?intention) ".")


3: (and (is-current-task orientation-interface)
        (program-step ? (? turn ?value) missing-step
                      ?intention))

   =>

   ("Another TURN command is necessary to orient "
          *TASK-NAME* " correctly.")

   ("TURN " ?value " is the setup step necessary to orient "
          *TASK-NAME* " correctly.")


4: (and (not (is-current-task orientation-interface))
        (program-step ? (? turn ?value) missing-step
                      ?intention))

   =>
```

```
("You are missing the rotation to turn through angle "
        (first-node ?intention) (second-node ?intention)
        (third-node ?intention) ".")

("You are missing the step TURN " ?value ".")
```

Hints for extra steps:

```
1: (and (is-current-task orientation-interface)
        (program-step ? (? turn ?value) extra-step))

   =>

   ("Only one TURN command is needed to set up the
        orientation correctly.")

   ("The step TURN " ?value " is unnecessary.")


2: (and (not (is-current-task orientation-interface))
        (program-step ? (? turn ?value) extra-step))

   =>

   ("There is an extra TURN command in your interface.")

   ("The step TURN " ?value " is unnecessary.")


3: (and (is-current-task orientation-interface)
        (program-step ? (? move ?value) extra-step))

   =>

   ("MOVE commands are not needed in the orientation
        interface.")

   ("The step MOVE " ?value " is unnecessary.")


4: (and (not (is-current-task orientation-interface))
        (program-step ? (? move ?value) extra-step))

   =>

   ("There is an extra MOVE command in your interface.")

   ("The step MOVE " ?value " is unnecessary.")
```

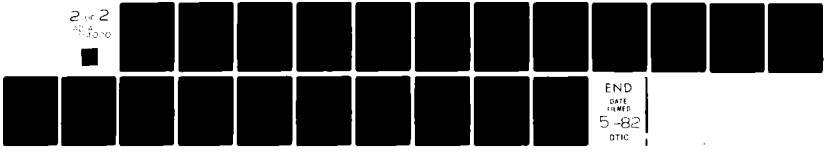
Hints for incorrect steps:

```
1: (and (is-current-task orientation-interface)
        (is-missing-step turn)
        (program-step ? ?step incorrect
```

```
                    (should-be ? ?value) ?intention))

   =>

   ("The setup rotation for " *TASK-NAME* " is incorrect.")

   ("TURN " ?value ", not TURN " (parameter ?step)
          " should be the setup for " *TASK-NAME*)


2: (and (not (is-current-task orientation-interface))
        (is-missing-step turn)
        (program-step ? ?step incorrect
                    (should-be ? ?value) ?intention))

   =>

   ("The rotation to turn through angle "
           (first-node ?intention) (second-node ?intention)
           (third-node ?intention) " is incorrect.")

   ("The rotation to turn through angle "
           (first-node ?intention) (second-node ?intention)
           (third-node ?intention) " should be TURN
           " ?value " instead of TURN " (parameter ?step)
           ".")


3: (and (is-missing-step move)
        (program-step ? ?step incorrect
                    (should-be ? ?value) ?intention))

   =>

   ("The retrace to move over line segment "
           (start-node ?intention) (end-node ?intention)
           " is incorrect.")

   ("The retrace to move over line segment "
           (start-node ?intention) (end-node ?intention)
           " should be MOVE " ?value " instead of MOVE "
           (parameter ?step) ".")
```

AD-A114 020 TEXAS INSTRUMENTS INC DALLAS CENTRAL RESEARCH LABS F/G 5/9
INTELLIGENT TUTORING FOR PROGRAMMING TASKS: USING PLAN ANALYSIS--ETC(U)
MAR 82 J R MILLER, T P KEHLER, P R MICHAELIS N00014-80-C-0818
UNCLASSIFIED TI-08-82-010 ONR-TR-82-0818F NL

2 OF 2
AD A 114020



END
DATE FILMED
5-82
DTIC

```
(PROGRAM-STEP ?
              (? DRAW ?VALUE)
              MISSING-STEP
              ?INTENTION)
```

Position variables are at the heart of this match: "?" matches anything, "?VALUE" matches any numerical value (such as the extent of a DRAW or a TURN), and "?INTENTION" matches any intended-action pattern, such as (DRAW-LINE-SEGMENT C A). Each of the hint generation structures is associated with two hints, one general and one specific:

```
General hint:
        ("You are missing the line segment from "
                (START-NODE ?INTENTION)
                " to "
                (END-NODE ?INTENTION)
                "." )

Specific hint:
        ("You are missing the step DRAW "
                ?VALUE
                " to draw the line segment from "
                (START-NODE ?INTENTION)
                " to "
                (END-NODE ?INTENTION)
                "." )
```

Presenting a hint to the student first requires determining whether the student should receive the general or the specific hint. When a student first asks for a hint, TURTLE presents the general hint; the specific hint is given in response to the second request; no hints are given on subsequent requests. The selected hint's pattern is then converted to the form the student will actually see by merging the text in the selected hint's pattern with the values of the variables and the function calls embedded within that pattern. In the present example, the following forms would be created:

General hint:
You are missing the line segment from C to A.

Specific hint:
You are missing the step DRAW 200 to draw the line segment from C to A.

These program-step structures are also used to control the student's correction of his program. After the student has defined his program and TURTLE has carried out the analysis described above, the student enters a debugging phase, where he is asked to identify and correct errors in his program (see the trace in Section 4.1.1). When a student attempts to correct one of these errors, a program-step structure is generated for the modification. If this structure indicates the modification is incorrect, TURTLE rejects the modification and tells him to try again. Correct steps cannot be deleted, and incorrect steps -- typically TURNs with the wrong angle specified -- must be respecified with the correct value. In this way, the student is kept working toward a correct solution at all times, although at the cost of preventing the student from experimenting with different, possibly incorrect, versions of his program.

4.3 Areas of Future Development

This research was focused on developing TURTLE's ability to identify the plans underlying students' LOGO programs and to offer hints that address errors in these plans. This initial effort has been limited in two ways, each of which is an important area for future research.

The first of these is to remove the restrictions on the kinds of LOGO programs that can be written; in particular, to

allow programs with variable assignment, recursion, and iteration. Fortunately, these components of LOGO can be incorporated into TURTLE's existing system of task networks by modifying and extending the CONNECTIONS section of a figure's task network. Consider the recursive function "BINARY-TREE LENGTH DEPTH", which would draw a binary tree DEPTH levels deep, with branches LENGTH units long. The task network for this figure would state that, while some of the figure's points are connected by MOVEs and DRAWs, others are connected by (recursive) calls to the BINARY-TREE function. In evaluating a student's program for this figure, TURTLE would have to determine that the student's program called BINARY-TREE at the appropriate places, and that the arguments defining the length and number of the branches of the sub-tree were specified correctly. This process would be no different from the way TURTLE currently determines that, for instance, an angle is generated by a TURN of a particular number of degrees; such an extension is completely compatible with TURTLE's existing representational and analysis by synthesis systems.

Other extensions of TURTLE -- ones which reach beyond the domain of turtle graphics -- are possible. However, one should not overlook the advantages of working within a highly constrained domain such as turtle graphics. TURTLE is successful largely because of the detailed problem representation found in the system's task networks; these make possible the generation of a set of possible solutions, one of which should correspond to the plan underlying the student's program. The success of future applications of TURTLE's methodology will depend upon the

presence of a comparably detailed specification of the new domain's tasks.

The second area in need of development is the construction of a reasonable pedagogical component for TURTLE, one that can develop a sound model of the student and use this model to control the tutorial process. TURTLE currently lacks a well-developed curriculum system such as that in BIP (Barr, et al., 1976), in which the tasks presented to students are selected on the basis of the skills the student has and has not yet acquired; the task selection strategies used by BIP would provide a sound beginning for this work. There is also much room for improvement in TURTLE's hinting capability. This project did not address the questions of when a student should be interrupted with a hint, and TURTLE has only primitive strategies for determining the exact hint a student should receive: TURTLE gives a general hint when a hint is first requested, and gives more specific hints in response to further requests. The nature of these problems is discussed in the summary; for now, it should be noted that these insufficiencies can be corrected only by the development of powerful student models, which will itself depend upon careful research into the psychological issues underlying the question of how people learn to program.

Section 5

Summary

The results of this research are encouraging with respect to the overall feasibility of constructing intelligent tutorial systems. TURTLE can exploit the limited structure of (a modified version of) LOGO and the programming problems presented to students to generate hints relevant to a student's proposed solution. We have thus far run only informal experiments with TURTLE, which have not been suitable for a detailed comparison of TURTLE to a more traditional tutor such as BIP. However, TURTLE, like the human tutors in the BIP studies, can identify the errors in a student's program and offer specific design-level and code-level hints relevant to these errors. In addition, TURTLE's use of menus and multiple windows corrects some of the human engineering problems observed in the BIP/HINT experiments.

As is often the case, however, the results of this research may raise more questions than they answer. In particular:

* What strategies and knowledge structures are acquired by people learning to program? The LOGO programs written by our subjects typically went through two stages. First, the primary figures (triangles, squares, etc.) were drawn correctly, but were interfaced incorrectly; second, the interfaces among these figures were gradually corrected. These studies suggest that people have good strategies for drawing regular geometric figures, but not for building the interfaces between figures.

Our subjects' major difficulty in drawing regular figures

was learning to specify the figures' angles by turning the turtle through some number of degrees. Some of these constructions were more difficult to learn than others: a 90 degree right turn is made by "TURN 90", while a 90 degree left turn is made by "TURN 270"; an equilateral triangle (with equal interior angles of 60 degrees) is generated by successively drawing the triangle's exterior 120 degree angles. These were not major problems for our students, however. After working through the TRIANGLE task in TURTLE, they were almost always able to correctly build the square and triangular components of the more complex figures taught by TURTLE, such as WISHING-WELL and NAPOLEON. In contrast, students had considerable difficulty building interfaces that interconnect the regular components of these complex figures. Most of these interfaces were initially incorrect, and most of the time spent by a student in a session with TURTLE was devoted to correcting interfaces. It is clear that, as people become more familiar with LOGO and turtle graphics, they become more able to build correct interfaces, and the study of the development of successful interface strategies may offer some valuable insights into computer programming as a cognitive process.

What constitutes a good hint? When a tutor has decided that a student needs a hint, that hint should

* address the immediate problem the student is having, and

* create in the student a strong conceptual understanding of the problem's solution, so that future hints will be unnecessary.

These goals typically interfere with each other. The student's immediate problem can best be solved by simply giving the student a set of program statements that will produce the desired result; if the student simply copies the provided solution, however, it is doubtful he will retain this advice for use at a later time. At the other extreme, if the tutor gives a vague, general hint about how the problem might be solved, the student will probably not acquire enough new information to allow him to either solve the immediate problem or increase his long-term understanding of the domain. A good tutorial system should provide hints that address the specific gaps and fallacies in the student's understanding of the problem. These errors may, of course, occur at any of the several levels of abstraction that characterize the student's problem representation, such as the multiple levels of design and code structures encountered in the programming tasks studied here.

This issue has received little study; it has been addressed in the work by Collins and his colleagues on Socratic tutorial systems (Collins, Aiello, Warnock, & Miller, 1975; Stevens & Collins, 1977). Most existing systems try to deal with this problem by providing two levels of hints (see the discussions of the OGOL tutor, BIP, and TURTLE): a general hint is given first (in these programming language tutors, these are often related to the design of the program or individual statements), followed by a more specific hint (often showing program statements similar to or the same as those that will solve the problem). This technique only approximates a tutor that can target the specificity of its hints to the needs of the student. Since such

a system does not always offer hints that address the exact nature of a student's problem, it forces the student who needs fairly specific help on the form of particular statements to ask for several hints, many of which may provide irrelevant or even confusing information. More significantly, students can become aware of this aspect of the tutor very quickly (as did some of the subjects in our informal experiments with TURTLE), and realize that the exact answers to their problems can be obtained simply by asking for several hints. Some students may then deliberately ask for multiple hints in order to be told the answer to their problem, while others who view asking for hints as admitting failure (see Section 3.4.4) may avoid hints altogether for fear of being given a hint that reveals the exact answer. Being able to generate a truly appropriate hint for a particular student in a particular task is thus a critical component of a truly successful tutor, one that will depend upon understanding the psychological components of the instructional and learning experiences, and incorporating these components into an accurate model of the student and the tutorial environment.

References

Atwood, M. E., & Jeffries, R. Studies in plan construction: Analysis of an extended protocol. Technical Report SAI-80-028-DEN, Englewood, Colorado: Science Applications, Inc., 1980.

Barr, A., Beard M., & Atkinson, R. The computer as a tutorial laboratory. International Journal of Man-Machine Studies, 1976, 8, 567-596.

Brown, J. S., & Burton, R. Multiple representations of knowledge for tutorial reasoning. In D. Bobrow & A. Collins (Eds.), Representation and understanding. New York: Academic Press, 1975.

Clancey, W. J., Bennett, J. S., & Cohen, P. R. Applications-oriented AI research: Education. In A. Barr & E. A. Feigenbaum (eds.), Handbook of artificial intelligence. Los Altos: William Kaufmann, 1981.

Gentner, D. The FLOW tutor: A schema-based tutorial system. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, 1977, 787-788.

Gentner, D., & Norman, D. A. The FLOW tutor: Schemas for tutoring. Technical Report 77-02, La Jolla, California: University of California at San Diego Center for Human Information Processing, 1977.

Goldstein, I. P. Summary of MYCROFT: A system for understanding simple picture programs. Artificial Intelligence,

1975, 6, 249-288.

Miller, M. L. A structured planning and debugging environment for elementary programming. International Journal of Man-Machine Studies, 1979, 11, 79-95.

Miller, M. L., & Goldstein, I. P. Structured planning and debugging. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, 1977.

Papert, S. Mindstorms. New York, Basic Books, 1980.

Sleeman, D., & Brown, J. S. Intelligent tutoring systems. New York: Academic Press, 1981.

Stevens, A. L., & Collins, A. The goal structure of a socratic tutor. Technical Report 3518, Bolt Beranek and Newman, Inc., 1977.

Appendix A:

TURTLE's implementation

TURTLE is implemented in ASSERT,[12] a CONNIVER-like assertional database system implemented in MACLISP. It uses a standard computer terminal for input and the display of TURTLE's program analysis, and a high resolution color graphics terminal (512 x 512 pixels) for the display of LOGO figures.

During the task solution and assistance process, TURTLE uses two graphics windows and one text window on the graphics terminal. The graphics windows display the figure a correct program should draw and the figure drawn by the student's current program. These two windows occupy the left half of the screen on the graphics terminal. A text window on the right half of the screen window displays program definitions and modifications, hints, and tutorial text. TURTLE can be run without a graphics terminal, although a diagram of the four tasks will be required to follow TURTLE's operation.

TURTLE's response time varies, but is usually under 3 seconds for requests for hints, program edits, text display, and similar operations. Open coded solutions can take more time, depending on the complexity of the figure to be analyzed (an open coded solution for the tree task with one or zero retraces can be recognized in 15 seconds or less).

---

[12] ASSERT was designed by Mark Miller and William Murray, and implemented by Murray.

Navy

1 Dr. Ed Aiken
Navy Personnel R&D Center
San Diego, CA 92152

1 Meryl S. Baker
NPRDC
Code P309
San Diego, CA 92152

1 Dr. Robert Breaux
Code N-711
NAVTRAEQUIPCEN
Orlando, FL 32813

1 CDR Mike Curran
Office of Naval Research
800 N. Quincy St.
Code 270
Arlington, VA 22217

1 DR. PAT FEDERICO
NAVY PERSONNEL R&D CENTER
SAN DIEGO, CA 92152

1 Dr. John Ford
Navy Personnel R&D Center
San Diego, CA 92152

1 LT Steven D. Harris, MSC, USN
Code 6021
Naval Air Development Center
Warminster, Pennsylvania 18974

1 Dr. Jim Hollan
Code 304
Navy Personnel R & D Center
San Diego, CA 92152

1 Dr. Norman J. Kerr
Chief of Naval Technical Training
Naval Air Station Memphis (75)
Millington, TN 38054

1 Dr. William L. Maloy
Principal Civilian Advisor for
Education and Training
Naval Training Command, Code 00A
Pensacola, FL 32508

Navy

1 CAPT Richard L. Martin, USN
Prospective Commanding Officer
USS Carl Vinson (CVN-70)
Newport News Shipbuilding and Drydock Co
Newport News, VA 23607

1 Dr. James McBride
Navy Personnel R&D Center
San Diego, CA 92152

1 Dr William Montague
Navy Personnel R&D Center
San Diego, CA 92152

1 Ted M. I. Yellen
Technical Information Office, Code 201
NAVY PERSONNEL R&D CENTER
SAN DIEGO, CA 92152

1 Library, Code P201L
Navy Personnel R&D Center
San Diego, CA 92152

1 Technical Director
Navy Personnel R&D Center
San Diego, CA 92152

6 Commanding Officer
Naval Research Laboratory
Code 2627
Washington, DC 20390

1 Psychologist
ONR Branch Office
Bldg 114, Section D
666 Summer Street
Boston, MA 02210

1 Office of Naval Research
Code 437
800 N. Quincy SStreet
Arlington, VA 22217

5 Personnel & Training Research Programs
(Code 458)
Office of Naval Research
Arlington, VA 22217

Navy

1 Psychologist
ONR Branch Office
1030 East Green Street
Pasadena, CA 91101

1 Special Asst. for Education and Training (OP-01E)
Rm. 2705 Arlington Annex
Washington, DC 20370

1 Office of the Chief of Naval Operations
Research Development & Studies Branch (OP-115)
Washington, DC 20350

1 LT Frank C. Petho, MSC, USN (Ph.D)
Selection and Training Research Division
Human Performance Sciences Dept.
Naval Aerospace Medical Research Laborat
Pensacola, FL 32508

1 Dr. Gary Poock
Operations Research Department
Code 55PK
Naval Postgraduate School
Monterey, CA 93940

1 Roger W. Remington, Ph.D
Code L52
NAMRL
Pensacola, FL 32508

1 Dr. Bernard Rimland (03B)
Navy Personnel R&D Center
San Diego, CA 92152

1 Dr. Worth Scanland, Director
Research, Development, Test & Evaluation
N-5
Naval Education and Training Command
NAS, Pensacola, FL 32508

1 Dr. Robert G. Smith
Office of Chief of Naval Operations
OP-987H
Washington, DC 20350

Navy

1 Dr. Alfred F. Smode
Training Analysis & Evaluation Group (TAEG)
Dept. of the Navy
Orlando, FL 32813

1 Dr. Richard Sorensen
Navy Personnel R&D Center
San Diego, CA 92152

1 Roger Weissinger-Baylon
Department of Administrative Sciences
Naval Postgraduate School
Monterey, CA 93940

1 Dr. Robert Wisher
Code 309
Navy Personnel R&D Center
San Diego, CA 92152

1 Mr John H. Wolfe
Code P310
U. S. Navy Personnel Research and Development Center
San Diego, CA 92152

Army

1 Technical Director
U. S. Army Research Institute for the Behavioral and Social Sciences
5001 Eisenhower Avenue
Alexandria, VA 22333

1 Mr. James Baker
Systems Manning Technical Area
Army Research Institute
5001 Eisenhower Ave.
Alexandria, VA 22333

1 Dr. Beatrice J. Farr
U. S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

1 DR. FRANK J. HARRIS
U.S. ARMY RESEARCH INSTITUTE
5001 EISENHOWER AVENUE
ALEXANDRIA, VA 22333

1 Dr. Michael Kaplan
U.S. ARMY RESEARCH INSTITUTE
5001 EISENHOWER AVENUE
ALEXANDRIA, VA 22333

1 Dr. Milton S. Katz
Training Technical Area
U.S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

1 Dr. Harold F. O'Neil, Jr.
Attn: PERI-OK
Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

1 Dr. Robert Sasmor
U. S. Army Research Institute for the Behavioral and Social Sciences
5001 Eisenhower Avenue
Alexandria, VA 22333

Army

1 Dr. Frederick Steinheiser
Dept. of Navy
Chief of Naval Operations
OP-113
Washington, DC 20350

1 Dr. Joseph Ward
U.S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Air Force

1 U.S. Air Force Office of Scientific Research
Life Sciences Directorate, NL
Bolling Air Force Base
Washington, DC 20332

1 Dr. Genevieve Haddad
Program Manager
Life Sciences Directorate
AFOSR
Bolling AFB, DC 20332

2 3700 TCHTW/TTGH Stop 32
Sheppard AFB, TX 76311

Marines

1 H. William Greenup
Education Advisor (E031)
Education Center, MCDEC
Quantico, VA 22134

1 Special Assistant for Marine Corps Matters
Code 100M
Office of Naval Research
800 N. Quincy St.
Arlington, VA 22217

1 DR. A.L. SLAFKOSKY
SCIENTIFIC ADVISOR (CODE RD-1)
HQ, U.S. MARINE CORPS
WASHINGTON, DC 20380

CoastGuard

1 Chief, Psychological Reserch Branch
U. S. Coast Guard (G-P-1/2/TP42)
Washington, DC 20593

Other DoD

12 Defense Technical Information Center
Cameron Station, Bldg 5
Alexandria, VA 22314
Attn: TC

1 Military Assistant for Training and
Personnel Technology
Office of the Under Secretary of Defense
for Research & Engineering
Room 3D129, The Pentagon
Washington, DC 20301

1 DARPA
1400 Wilson Blvd.
Arlington, VA 22209

Civil Govt

1 Dr. Susan Chipman
Learning and Development
National Institute of Education
1200 19th Street NW
Washington, DC 20208

1 Dr. John Mays
National Institute of Education
1200 19th Street NW
Washington, DC 20208

1 William J. McLaurin
66610 Howie Court
Camp Springs, MD 20031

1 Dr. Arthur Melmed
National Intitute of Education
1200 19th Street NW
Washington, DC 20208

1 Dr. Andrew R. Molnar
Science Education Dev.
and Research
National Science Foundation
Washington, DC 20550

1 Dr. Joseph Psotka
National Institute of Education
1200 19th St. NW
Washington,DC 20208

1 Dr. Frank Withrow
U. S. Office of Education
400 Maryland Ave. SW
Washington, DC 20202

1 Dr. Joseph L. Young, Director
Memory & Cognitive Processes
National Science Foundation
Washington, DC 20550

Non Govt

1 Dr. John R. Anderson
Department of Psychology
Carnegie Mellon University
Pittsburgh, PA 15213

1 Anderson, Thomas H., Ph.D.
Center for the Study of Reading
174 Children's Research Center
51 Gerty Drive
Champiagn, IL 61820

1 Dr. John Annett
Department of Psychology
University of Warwick
Coventry CV4 7AL
ENGLAND

1 1 psychological research unit
Dept. of Defense (Army Office)
Campbell Park Offices
Canberra ACT 2600, Australia

1 Dr. Alan Baddeley
Medical Research Council
Applied Psychology Unit
15 Chaucer Road
Cambridge CB2 2EF
ENGLAND

1 Dr. Patricia Baggett
Department of Psychology
University of Colorado
Boulder, CO 80309

1 Mr Avron Barr
Department of Computer Science
Stanford University
Stanford, CA 94305

1 Liaison Scientists
Office of Naval Research,
Branch Office , London
Box 39 FPO New York 09510

1 Dr. Lyle Bourne
Department of Psychology
University of Colorado
Boulder, CO 80309

Non Govt

1 Dr. John S. Brown
XEROX Palo Alto Research Center
3333 Coyote Road
Palo Alto, CA 94304

1 Dr. Bruce Buchanan
Department of Computer Science
Stanford University
Stanford, CA 94305

1 DR. C. VICTOR BUNDERSON
WICAT INC.
UNIVERSITY PLAZA, SUITE 10
1160 SO. STATE ST.
OREM, UT 84057

1 Dr. Pat Carpenter
Department of Psychology
Carnegie-Mellon University
Pittsburgh, PA 15213

1 Dr. John B. Carroll
Psychometric Lab
Univ. of No. Carolina
Davie Hall 013A
Chapel Hill, NC 27514

1 Dr. William Chase
Department of Psychology
Carnegie Mellon University
Pittsburgh, PA 15213

1 Dr. Micheline Chi
Learning R & D Center
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15213

1 Dr. Allan M. Collins
Bolt Beranek & Newman, Inc.
50 Moulton Street
Cambridge, Ma 02138

1 Dr. Lynn A. Cooper
LRDC
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15213

Non Govt

1 Dr. Meredith P. Crawford
American Psychological Association
1200 17th Street, N.W.
Washington, DC 20036

1 Dr. Kenneth B. Cross
Anacapa Sciences, Inc.
P.O. Drawer Q
Santa Barbara, CA 93102

1 LCOL J. C. Eggenberger
DIRECTORATE OF PERSONNEL APPLIED RESEARC
NATIONAL DEFENCE HQ
101 COLONEL BY DRIVE
OTTAWA, CANADA K1A OK2

1 Dr. Ed Feigenbaum
Department of Computer Science
Stanford University
Stanford, CA 94305

1 Dr. Richard L. Ferguson
The American College Testing Program
P.O. Box 168
Iowa City, IA 52240

1 Mr. Wallace Feurzeig
Bolt Beranek & Newman, Inc.
50 Moulton St.
Cambridge, MA 02138

1 Dr. Victor Fields
Dept. of Psychology
Montgomery College
Rockville, MD 20850

1 Univ. Prof. Dr. Gerhard Fischer
Liebiggasse 5/3
A 1010 Vienna
AUSTRIA

1 DR. JOHN D. FOLLEY JR.
APPLIED SCIENCES ASSOCIATES INC
VALENCIA, PA 16059

1 Dr. John R. Frederiksen
Bolt Beranek & Newman
50 Moulton Street
Cambridge, MA 02138

Non Govt

1 Dr. Alinda Friedman
Department of Psychology
University of Alberta
Edmonton, Alberta
CANADA T6G 2E9

1 DR. ROBERT GLASER
LRDC
UNIVERSITY OF PITTSBURGH
3939 O'HARA STREET
PITTSBURGH, PA 15213

1 Dr. Marvin D. Glock
217 Stone Hall
Cornell University
Ithaca, NY 14853

1 Dr. Daniel Gopher
Industrial & Management Engineering
Technion-Israel Institute of Technology
Haifa
ISRAEL

1 DR. JAMES G. GREENO
LRDC
UNIVERSITY OF PITTSBURGH
3939 O'HARA STREET
PITTSBURGH, PA 15213

1 Dr. Ron Hambleton
School of Education
University of Massechusetts
Amherst, MA 01002

1 Dr. Harold Hawkins
Department of Psychology
University of Oregon
Eugene OR 97403

1 Dr. Barbara Hayes-Roth
The Rand Corporation
1700 Main Street
Santa Monica, CA 90406

1 Dr. Frederick Hayes-Roth
The Rand Corporation
1700 Main Street
Santa Monica, CA 90406

Non Govt

1 Dr. James R. Hoffman
Department of Psychology
University of Delaware
Newark, DE 19711

1 Dr. Kristina Hooper
Clark Kerr Hall
University of California
Santa Cruz, CA 95060

1 Glenda Greenwald, Ed.
"Human Intelligence Newsletter"
P. O. Box 1163
Birmingham, MI 48012

1 Dr. Earl Hunt
Dept. of Psychology
University of Washington
Seattle, WA 98105

1 Dr. Ed Hutchins
Navy Personnel R&D Center
San Diego, CA 92152

1 DR. KAY INABA
21116 VANOWEN ST
CANOGA PARK, CA 91303

1 Dr. Steven W. Keele
Dept. of Psychology
University of Oregon
Eugene, OR 97403

1 Dr. Walter Kintsch
Department of Psychology
University of Colorado
Boulder, CO 80302

1 Dr. David Kieras
Department of Psychology
University of Arizona
Tuscon, AZ 85721

1 Dr. Stephen Kosslyn
Harvard University
Department of Psychology
33 Kirkland Street
Cambridge, MA 02138

Non Govt

1 Dr. Marcy Lansman
Department of Psychology, NI 25
University of Washington
Seattle, WA 98195

1 Dr. Jill Larkin
Department of Psychology
Carnegie Mellon University
Pittsburgh, PA 15213

1 Dr. Alan Lesgold
Learning R&D Center
University of Pittsburgh
Pittsburgh, PA 15260

1 Dr. Michael Levine
Department of Educational Psychology
210 Education Bldg.
University of Illinois
Champaign, IL 61801

1 Dr. Robert Linn
College of Education
University of Illinois
Urbana, IL 61801

1 Dr. Erik McWilliams
Science Education Dev. and Research
National Science Foundation
Washington, DC 20550

1 Dr. Mark Miller
TI Computer Science Lab
C/O 2824 Winterplace Circle
Plano, TX 75075

1 Dr. Allen Munro
Behavioral Technology Laboratories
1845 Elena Ave., Fourth Floor
Redondo Beach, CA 90277

1 Dr. Donald A Norman
Dept. of Psychology C-009
Univ. of California, San Diego
La Jolla, CA 92093

Non Govt

1 Committee on Human Factors
JH 811
2101 Constitution Ave. NW
Washington, DC 20418

1 Dr. Jesse Orlansky
Institute for Defense Analyses
400 Army Navy Drive
Arlington, VA 22202

1 Dr. Seymour A. Papert
Massachusetts Institute of Technology
Artificial Intelligence Lab
545 Technology Square
Cambridge, MA 02139

1 Dr. James A. Paulson
Portland State University
P.O. Box 751
Portland, OR 97207

1 Dr. James W. Pellegrino
University of California,
Santa Barbara
Dept. of Psychology
Santa Barabara, CA 93106

1 MR. LUIGI PETRULLO
2431 N. EDGEWOOD STREET
ARLINGTON, VA 22207

1 Dr. Martha Polson
Department of Psychology
Campus Box 346
University of Colorado
Boulder, CO 80309

1 DR. PETER POLSON
DEPT. OF PSYCHOLOGY
UNIVERSITY OF COLORADO
BOULDER, CO 80309

1 Dr. Steven E. Poltrock
Department of Psychology
University of Denver
Denver,CO 80208

Non Govt

1 MINRAT M. L. RAUCH
P II 4
BUNDESMINISTERIUM DER VERTEIDIGUNG
POSTFACH 1328
D-53 BONN 1, GERMANY

1 Dr. Fred Reif
SESAME
c/o Physics Department
University of California
Berkely, CA 94720

1 Dr. Lauren Resnick
LRDC
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15213

1 Mary Riley
LRDC
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15213

1 Dr. Andrew M. Rose
American Institutes for Research
1055 Thomas Jefferson St. NW
Washington, DC 20007

1 Dr. Ernst Z. Rothkopf
Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

1 Dr. David Rumelhart
Center for Human Information Processing
Univ. of California, San Diego
La Jolla, CA 92093

1 DR. WALTER SCHNEIDER
DEPT. OF PSYCHOLOGY
UNIVERSITY OF ILLINOIS
CHAMPAIGN, IL 61820

1 Dr. Alan Schoenfeld
Department of Mathematics
Hamilton College
Clinton, NY 13323

Non Govt

1 DR. ROBERT J. SEIDEL
INSTRUCTIONAL TECHNOLOGY GROUP
HUMRRO
300 N. WASHINGTON ST.
ALEXANDRIA, VA 22314

1 Committee on Cognitive Research
% Dr. Lonnie R. Sherrod
Social Science Research Council
605 Third Avenue
New York, NY 10016

1 Robert S. Siegler
Associate Professor
Carnegie-Mellon University
Department of Psychology
Schenley Park
Pittsburgh, PA 15213

1 Dr. Edward E. Smith
Bolt Beranek & Newman, Inc.
50 Moulton Street
Cambridge, MA 02138

1 Dr. Robert Smith
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903

1 Dr. Richard Snow
School of Education
Stanford University
Stanford, CA 94305

1 Dr. Robert Sternberg
Dept. of Psychology
Yale University
Box 11A, Yale Station
New Haven, CT 06520

1 DR. ALBERT STEVENS
BOLT BERANEK & NEWMAN, INC.
50 MOULTON STREET
CAMBRIDGE, MA 02138

1 David E. Stone, Ph.D.
Hazeltine Corporation
7680 Old Springhouse Road
McLean, VA 22102

Non Govt

1 DR. PATRICK SUPPES
INSTITUTE FOR MATHEMATICAL STUDIES IN THE SOCIAL SCIENCES
STANFORD UNIVERSITY
STANFORD, CA 94305

1 Dr. Kikumi Tatsuoka
Computer Based Education Research Laboratory
252 Engineering Research Laboratory
University of Illinois
Urbana, IL 61801

1 Dr. John Thomas
IBM Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

1 DR. PERRY THORNDYKE
THE RAND CORPORATION
1700 MAIN STREET
SANTA MONICA, CA 90406

1 Dr. Douglas Towne
Univ. of So. California
Behavioral Technology Labs
1845 S. Elena Ave.
Redondo Beach, CA 90277

1 Dr. J. Uhlaner
Perceptronics, Inc.
6271 Variel Avenue
Woodland Hills, CA 91364

1 Dr. Benton J. Underwood
Dept. of Psychology
Northwestern University
Evanston, IL 60201

1 Dr. David J. Weiss
N660 Elliott Hall
University of Minnesota
75 E. River Road
Minneapolis, MN 55455

1 DR. GERSHON WELTMAN
PERCEPTRONICS INC.
6271 VARIEL AVE.
WOODLAND HILLS, CA 91367

Non Govt

1 Dr. Keith T. Wescourt
Information Sciences Dept.
The Rand Corporation
1700 Main St.
Santa Monica, CA 90406